SAPIENZA
UNIVERSITÀ DI ROMA

Automata-Theoretic Techniques for Reasoning and Learning in Linear-Time Temporal Logics on Finite Traces

Department of Computer, Control and Management Engineering

Dottorato di Ricerca in Engineering in Computer Science – XXXIV Ciclo

Candidate

Marco Favorito

ID number 1609890

Thesis Advisor

Prof. Giuseppe De Giacomo

Co-Advisor

Prof. Luca Iocchi

2021/2022

Thesis defended on the 26th of September
in front of a Board of Examiners composed by:

Prof. Giancarlo Fortino (chairman)

Prof. Fabrizio Maria Maggi

Prof. Luca Cabibbo

External reviewers:

Prof. Aniello Murano

Prof. Marco Roveri

Automata-Theoretic Techniques for Reasoning and Learning in Linear-Time Temporal Logics on Finite Traces

Ph.D. thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Website: https://marcofavorito.me

Author's email: favorito@diag.uniroma1.it

Abstract

The use of temporal logics on finite traces, like Linear Temporal Logic (LTL$_f$) and Linear Dynamic Logic (LDL$_f$), has shown to be very powerful for AI. In particular, they have been successfully applied in several AI fields, such as temporal synthesis, FOND planning, the theory of Markov Decision Processes, Reinforcement Learning, and Business Process Management. Almost all the techniques developed in recent years rely on the well-known connection between temporal logics and automata theory. In particular, the size of a deterministic finite automaton equivalent to an LTL$_f$/LDL$_f$ formula is, in the worst-case, doubly-exponentially larger than the formula. Nevertheless, such transformation is much better behaved with respect to the infinite traces setting, and this opens new avenues for algorithms that work well in practice.

This thesis aims to take some of these avenues, and open new ones, both in the theory and the applications of temporal logics in AI.

As a first contribution, we present a novel compositional technique for transforming an LDL$_f$ formula into a minimal DFA, and propose an efficient symbolic implementation that is competitive with state-of-the-art tools. The impressive results obtained open new possibilities for further research on this direction, as well as a ready-to-use tool for several applications in AI.

Then, we studied new problems in applying temporal logics in the context of Reinforcement Learning and Markov Decision Processes. In particular, we study the novel problem of Restraining Bolts, in which an authority imposes a restraining specification, written in LTL$_f$/LDL$_f$, to the acting of a reinforcement learning agent. Despite the authority and the learning agent having different representations of the world, we can show that, under general circumstances, the agent can learn its goals to suitably conform (as much as possible) to the restraining bolt specifications. We also studied variants of this problem and methods to engineer restraining specifications in order to improve the learning process.

In the area of LTL$_f$ synthesis, we develop the theory and the implementation of a forward technique that, in many cases, is able to cope with the costly translation to automata by building the automaton on-the-fly. We drastically improve related works on the topic by using an AND/OR graph search algorithm and Knowledge Compilation techniques to efficiently explore the search graph. The experimental results are very promising. This contribution is the starting point for cross-fertilization between the Synthesis and Planning community, and in particular for the development of a science of heuristics for LTL$_f$ synthesis, as has happened in Planning.

Contents

Chapter 1

Introduction

This chapter presents the outline of this thesis and summarises its motivation, goals, and achievements. Section 1.1 provides a high-level introduction to applications of temporal logics for AI techniques in decision-making. Then, Section 1.2 summarizes the major contributions of the work which was done towards this thesis. The structure of the rest of this thesis is discussed in Section 1.3.

1.1 Context: Temporal Logics for AI in Decision-Making

1.1.1 Logic and Computer Science

The connection between Logic and Computer Science (CS) has been widely acknowledged in the academic community (Halpern et al., 2001; Davis, 2018; Davis, 1988; Manna and Waldinger, 1993; Gottlob, 2009). From a historical perspective, mathematical logic was developed in an attempt to solve the crisis in the foundations of mathematics that emerged at the beginning of the 20th century. Between 1900 and 1930, this development was led by the mathematician David Hilbert's Program, whose main aim was to formalize all of mathematics and establish that mathematics is *complete* and *decidable*. Informally, completeness means that all "true" mathematical statements can be "proved", whereas decidability means that there is a mechanical rule to determine whether a given mathematical statement is "true" or "false". Hilbert's belief was really that for any problem, it is always possible to prove, in a finite number of unambiguous steps (with what nowadays is called an *algorithm*), its truth or its untruth.

However, this belief has been refuted by Kurt Gödel in his celebrated paper (Gödel, 1931), where he showed that the standard first-order axioms of arithmetic were incomplete. Furthermore, Alan Turing, Alonzo Church, and Alfred Tarski demonstrated the undecidability of first-order logic. Specifically, the set of all valid first-order sentences was shown to be undecidable (Church, 1936; Turing et al., 1936) whereas the set of all first-order sentences that are true in arithmetic was shown to be highly undecidable (Tarski, 1936).

Since then, logic has permeated through computer science. In particular, computer science benefitted from logic in the areas of computational complexity (Garey and Johnson, 1979; Immerman, 1999; Papadimitriou, 2007), database theory (Abiteboul, Hull, and Vianu, 1995), programming languages (Reynolds, 1998), reasoning about knowledge (Fagin et al., 1995), and computer-aided verification (Clarke, Emerson, and Sistla, 1986; Lichtenstein and Pnueli, 1985; Queille and Sifakis, 1982; Vardi and Wolper, 1986), just to name a few.

1.1.2 Temporal Logics and Program Verification

Program Verification, in particular Program Synthesis, is one of such fields. It started from a Church's paper back in 1957 (Church, 1963), in which he described the use of logic to specify and verify sequential circuits. A sequential circuit is a switching circuit whose output depends not only upon its input but also on what its input has been in the past. It can be seen as a particular type of finite-state machine, which became a subject of study in mathematical logic and computer science in the 1950s. Informally, the synthesis problem is to come up with mechanical translation of human-understandable task specifications to a program that is known to meet the specifications (Church, 1964). Since then, the topic has received very much attention from the scientific community, but the "big bang" (to use Moshe Vardi's words (Vardi, 2008)) for the application of logic to program correctness occurred with Amir Pnueli's 1977 paper (Pnueli, 1977), by advocating using future linear temporal logic (LTL), a type of *temporal logic*, as a logic for the specification of non-terminating programs

What are temporal logics? Differently from classical logics, temporal logics allow us to reason about propositions in terms of time (e.g. "I am *always* hungry", "I will *eventually* be hungry", or "I will be hungry *until* I eat something"). The history of time in logic goes back to ancient times (see (Ohrstrom and Hasle, 2007) for a detailed history), but the birth of modern temporal logic is unquestionably credited to the philosopher Arthur Norman Prior (Prior, 2003). The connection between classical logics and temporal logics is due to Hans Kamp in his PhD Thesis, dated 1968 (Kamp, 1968), by essentially showing that monadic first-order logic over the ordered naturals and linear-time temporal logic were mutually reducible logics.

Back to LTL. Thanks to its declarativeness and human-friendliness, Linear-time Temporal Logic turned out to be particularly suited for the specification of *reactive systems* (Pnueli, 1985), i.e. systems that have to operate continuously, e.g. hardware, operating systems, communication protocols, robots, etc. Examples of interesting specifications in the context of a distributed system are:

- (mutual exclusion): two processes can never be simultaneously in a critical section;
- (conditional response): if a process requests a resource, then eventually the resource will be granted;
- (liveness): eventually the process terminates.

One of the major approaches to reactive synthesis is the *automata-theoretic approach* (Nagel, Suppes, and Tarski, 1966; Hopcroft, Motwani, and Ullman, 2006). The key idea underlying the automata-theoretic approach is that, given an LTL formula φ, it is possible to construct a finite-state automaton $\mathcal{A}$ on infinite words that accepts precisely all computations that satisfy φ (Vardi and Wolper, 1994).

The correspondence between logic and automata is well-known in the academic community, and the automata-theoretic approach is at the foundation of many other techniques that deal with temporal logics (Buchi and Landweber, 1969; Vardi and Wolper, 1986; Vardi, 2003; Vardi, 1995; Burch et al., 1992; Gerth et al., 1995).

1.1.3 Temporal Logics and Artificial Intelligence

The field of Artificial Intelligence (AI) (Russell and Norvig, 2010), and in particular the Knowledge Representation (KR) (Brachman and Levesque, 2004; Reiter, 2001)

and Planning (Ghallab, Nau, and Traverso, 2004; Geffner and Bonet, 2013) community, is well aware of temporal logics since a long time, as they have been used for temporal specification of the course of actions of an agent or a system of agents (Fagin et al., 1995). For example, in reasoning about actions and planning, LTL has often been used as a specification mechanism for temporally extended goals (Bacchus and Kabanza, 1996; Bacchus and Kabanza, 2000; Felli, De Giacomo, and Lomuscio, 2012; Patrizi et al., 2011), temporal constraints on trajectories (Gabaldon, 2004; Gerevini et al., 2009a), for expressing preferences and soft constraints (Bienvenu, Fritz, and McIlraith, 2006; Bienvenu, Fritz, and McIlraith, 2011; Sohrabi, Baier, and McIlraith, 2011), for specifying declarative control knowledge on trajectories (Baier and McIlraith, 2006), for procedural control knowledge on trajectories (Baier, Fritz, and McIlraith, 2007), for planning via model checking using CTL (Cimatti, Giunchiglia, et al., 1997) and LTL (De Giacomo and Vardi, 1999), for temporal specifications in planning domains (Calvanese, De Giacomo, and Vardi, 2002), for specifying non-Markovian reward functions in Non-Markovian Reward Decision Processes (Bacchus, Boutilier, and Grove, 1996; Thiébaux et al., 2006), and in Declarative Business Process Management (BPM) (Pesic and Aalst, 2006; Pesic, Bosnacki, and Aalst, 2010).

The above techniques all rely on LTL, which is a temporal logic whose semantic is over ω-traces, i.e. infinite words. However, especially in the context of temporal constraints and preferences, LTL formulas are used to express properties or constraints on *finite* traces of actions/states; in fact, this can be done even if the standard semantics of LTL is defined on infinite traces. Nevertheless, often, the distinction between interpreting LTL on infinite or finite traces is blurred (De Giacomo, Masellis, and Montali, 2014). In fact, this assumption has been considered a sort of accident in much of the AI and BPM literature, and standard temporal logics (on infinite traces) have been "hacked" to fit this assumption, with some success, but only lately, clean solutions have been devised. In reality, interpreting temporal constraints/goals on finite traces is different than interpreting them on infinite traces (and much more well-behaved). Moreover, in AI, almost always, the focus is actually on finite traces. For example, in planning, the agent has a task specification or "goal", and has to produce a "plan" to satisfy the task in the environment model. However, the task has to terminate (typically, just reaching a certain state in the environment, i.e. a reachability goal) because if it were not the case, the agent would be stuck into doing the same task forever. But then, why bother with equipping it with a model of the environment and of the task at all? The other motivation is practical. In problems like reactive synthesis of an LTL specification φ (which is on infinite traces), the classical automata-theoretic solution involves the following steps:

1. compute the corresponding Büchi Nondeterministic Automaton (NBW) (Nagel, Suppes, and Tarski, 1966),
2. determinize the NBW into a Deterministic Parity Automaton (DPW) (exponential in the number of states, polytime in the number of priorities),
3. synthesize a winning strategy for parity game (polytime in the number of states, exponential in the number of priorities).

For the determinization in step 2, no scalable algorithm exists yet. In fact, the problem is highly intractable: from a 9-state NBW, its DRW counterpart has 1,059,057-state DRW (Althoff, Thomas, and Wallmeier, 2005), and there are no symbolic algorithms for it. Moreover, in step 3, solving parity games requires computing nested fixpoints (possibly exponentially many). Despite, as we shall see,

the synthesis problem in the finite-trace setting is still 2EXPTIME-complete as in the infinite-trace setting, automata-theoretic algorithms for automata on finite traces are much more well-behaved wrt the ones on infinite traces, and so amenable for effective implementations.

1.1.4 Temporal Logics on Finite Traces

For these reasons, Linear Temporal Logic *on finite traces* (LTL$_f$) has been advocated in (De Giacomo and Vardi, 2013) as a proper variant of LTL interpreted over finite traces. Moreover, at no cost of computational complexity but higher expressive power, the authors propose a novel formalism, Linear Dynamic Logic on finite traces (LDL$_f$); it is as expressive as a regular expression, while retaining the declarative nature and intuitive appeal of LTL$_f$. As in the case of infinite traces, both LTL$_f$ and LDL$_f$ have a tight connection with (finite traces) automata theory. Indeed, for an LTL$_f$/LDL$_f$ formula φ, it is possible to compute an Alternating Finite Automaton (AFA) (Chandra, Kozen, and Stockmeyer, 1981; Brzozowski and Leiss, 1980; Leiss, 1981) $\mathcal{A}$ that accepts the traces that satisfy φ. This makes reasoning over LTL$_f$/LDL$_f$ very appealing wrt the infinite trace counterpart, as in finite-trace settings, algorithms over automata, e.g. determinization becomes doable in practice. Once determinized, the resulting Deterministic Finite Automaton (DFA) (Rabin and Scott, 1959) can be exploited to efficiently execute a run over the trace produced by the system and easily verify whether the system of interest is well-behaving wrt the original specification.

The new version of LTL$_f$ and its pure-past counterpart (De Giacomo, Di Stasio, et al., 2020) have been quite successful in the AI and Formal Methods communities in recent years. For example, they have been used for finite temporal synthesis (De Giacomo and Vardi, 2015; De Giacomo and Vardi, 2016; Camacho, Baier, et al., 2018; Zhu, Tabajara, Li, et al., 2017), in Fully-Observable Non-Deterministic (FOND) Planning for LTL$_f$ Goals (De Giacomo and Rubin, 2018; Brafman and De Giacomo, 2019b; Brafman and De Giacomo, 2019b; Camacho and McIlraith, 2019a; De Giacomo, Favorito, and Fuggitti, 2022), in the theory of Markov Decision Processes (MDP) to capture non-Markovian rewards (Gretton, 2014; Lacerda, Parker, and Hawes, 2015; Brafman, De Giacomo, and Patrizi, 2018; Brafman and De Giacomo, 2019a; De Giacomo, Favorito, Iocchi, Patrizi, and Ronca, 2020) with applications in reinforcement learning (RL) (Puterman, 1994; Sutton and Barto, 1998) with temporal specifications (Camacho, Icarte, et al., 2019; De Giacomo, Iocchi, et al., 2019), to specify and monitor business processes (De Giacomo, Masellis, and Montali, 2014; De Giacomo, Masellis, Grasso, et al., 2014; De Giacomo, De Masellis, et al., 2020), and many others.

For all these techniques, the foundational building block is the transformation from LTL$_f$/LDL$_f$ formulas into a DFA. Figure 1.1 depicts the workflow when working with LTL$_f$/LDL$_f$ applied to AI. The LTL$_f$/LDL$_f$ formula φ is first transformed into a DFA $\mathcal{A}$, and then combined with other techniques, depending on the field of application. We now briefly survey how this connection actually emerges in these domains.

FOND planning. Planning for LTL$_f$ goals has been studied in, e.g., (Baier and McIlraith, 2006; De Giacomo and Vardi, 2013; Torres and Baier, 2015) for deterministic domains and in (De Giacomo and Vardi, 2015; Camacho, Triantafillou, et al., 2017; De Giacomo and Rubin, 2018) for nondeterministic domain. The overall approach for FOND planning with LTL$_f$ goals (De Giacomo and Rubin, 2018) is to compute both the DFA of the formula and the DFA of the planning domain, compute

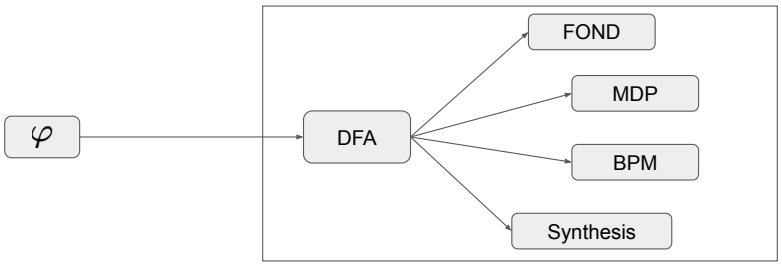


Figure 1.1. Diagram that shows how the LTL$_f$/LDL$_f$-to-DFA problem is at the foundations of other temporal logic-based areas in AI. An LTL$_f$/LDL$_f$ formula φ is usually translated into a DFA and then combined with other techniques from the areas of: Fully-Observable Non-Deterministic (FOND) planning; the theory of Markov Decision Processes (MDP); Business Process Management (BPM) and Finite Temporal Synthesis of LTL$_f$/LDL$_f$ specifications.

the product, and solve the DFA game (De Giacomo and Vardi, 2015) by returning a winning strategy if it exists.

Theory of MDPs. Specification of reward functions using temporal logics is known since (Bacchus, Boutilier, and Grove, 1996). But more recently, it has been done using LTL$_f$/LDL$_f$ for specifying the reward function and then planning with MDPs (Brafman, De Giacomo, and Patrizi, 2018). Given a temporal logic specification (φ, r), where φ is an LTL$_f$/LDL$_f$ formula and $r \in \mathbb{R}$ is a reward value, the approach first computes the automaton $\mathcal{A}_\varphi$, and then computes the synchronous product between $\mathcal{A}_\varphi$ and the Non-Markovian MDP $\mathcal{M}$, resulting in a new MDP with an extended state space, i.e. the original state plus the needed memory to let the agent remembering the salient information of the history so far, in function of the satisfaction of φ. Another interesting application is in Regular Decision Processes (Brafman and De Giacomo, 2019b), a novel model of the environment where not just the reward function but also the transition function is non-Markovian, i.e. the transition probabilities from a given state depend on past information but only "regularly" (i.e. with a function that is not more complex to evaluate than regular languages). Both the transitions and the rewards are specified by means of LTL$_f$/LDL$_f$ formulas, which can be further compiled into transducers (Moore, 1956; Mealy, 1955). In particular, such model has gained very recent interest in the reinforcement learning community (Gaon and Brafman, 2020; Abadi and Brafman, 2020; Ronca and De Giacomo, 2021; Ronca, Licks, and De Giacomo, 2022). Other related works in the related field of reinforcement learning with temporal logic specifications are (Icarte, Klassen, et al., 2018b; Camacho, Icarte, et al., 2019; Alshiekh et al., 2018; Aksaray et al., 2016; Littman et al., 2017; Hasanbeig, Abate,

and Kroening, 2019; Hasanbeig, Kantaros, et al., 2019; Jothimurugan et al., 2021; Li, Vasile, and Belta, 2017).

Business Process Management. LTL is at the base of one of the main declarative process modelling approaches: DECLARE (Pesic and Aalst, 2006; Montali et al., 2010; Maggi et al., 2011). Recently, DECLARE has been revised in terms of finite traces (De Giacomo, De Masellis, and Montali, 2014). In (De Giacomo, Maggi, et al., 2017), the problem of the LTL_f-based trace alignment has been studied, which is one major task in business process management where the aim is to align real process execution traces to a process model by (minimally) introducing and eliminating steps. It is, again, based on the translation from $\text{LTL}_f/\text{LDL}_f$ formulas to DFAs. In runtime monitoring (Aalst, 2011) which is one of the central tasks to provide operational decision support to running business processes, it can be checked on-the-fly whether they comply with constraints and rules specified in $\text{LTL}_f/\text{LDL}_f$ formulas (De Giacomo, De Masellis, et al., 2020; De Giacomo, Masellis, Grasso, et al., 2014; Maggi et al., 2011).

Finite Temporal Synthesis. The foundations for $\text{LTL}_f/\text{LDL}_f$ synthesis have been laid down in (De Giacomo and Vardi, 2015; De Giacomo and Vardi, 2016). The automata-based procedure for $\text{LTL}_f/\text{LDL}_f$ synthesis is as follows: (i) compute the DFA of the $\text{LTL}_f/\text{LDL}_f$ formula φ; (ii) find a winning strategy over the DFA game induced by the partition of controllable and uncontrollable variables. Since then, a plethora of extensions and variants have been studied: LTL_f synthesis under environment specifications (Aminof, De Giacomo, Murano, et al., 2019; Zhu, Giacomo, et al., 2020; Camacho, Bienvenu, and McIlraith, 2018; De Giacomo, Stasio, Vardi, et al., 2020; De Giacomo, Stasio, Tabajara, et al., 2021), LTL_f synthesis with mandatory stop actions (De Giacomo, Stasio, Perelli, et al., 2021), best-effort LTL_f synthesis (Aminof, De Giacomo, and Rubin, 2021; Aminof, De Giacomo, Lomuscio, et al., 2021; Aminof, De Giacomo, Lomuscio, et al., 2020; Ciolek et al., 2020), forward LTL_f synthesis (Xiao et al., 2021). Among implementations, we report *Syft* (Zhu, Tabajara, Li, et al., 2017), *Lisa* (Bansal et al., 2020), and *LTLfSyn* (Xiao et al., 2021).

The impressive energies the community has put on the use of $\text{LTL}_f/\text{LDL}_f$ formalisms in all of these AI problems stands on its own as a motivation of this thesis to exist.

1.1.5 $\text{LTL}_f/\text{LDL}_f$-to-DFA: State-of-the-art

As already stressed earlier, the crux of the applications of $\text{LTL}_f/\text{LDL}_f$ to the AI problems is the computation of the DFA which is semantically equivalent to a $\text{LTL}_f/\text{LDL}_f$ formula. In the worst case, if the formula φ is of size n, then the minimal DFA $\mathcal{A}_\varphi$ can have a state space whose size is doubly exponentially larger, i.e. 2^{2^n} (Chandra, Kozen, and Stockmeyer, 1981). This is a discouraging complexity result, although this does not mean that good algorithms and implementation cannot be found. In fact, practice has shown that many instances of $\text{LTL}_f/\text{LDL}_f$-to-DFA are tractable (Tabakov and Vardi, 2005), i.e. on average, we can compute the DFA of a formula $\text{LTL}_f/\text{LDL}_f$ in a reasonable time. Figure 1.2 gives an overview of state-of-the-art techniques and tools to compute the DFA from a LTL_f formula.

Another approach is to avoid the full construction of the DFA, and instead build the automaton *on-the-fly*, in a way that depends on the actual problem being solved. This idea has been suggested several times in the literature, e.g. (Brafman, De

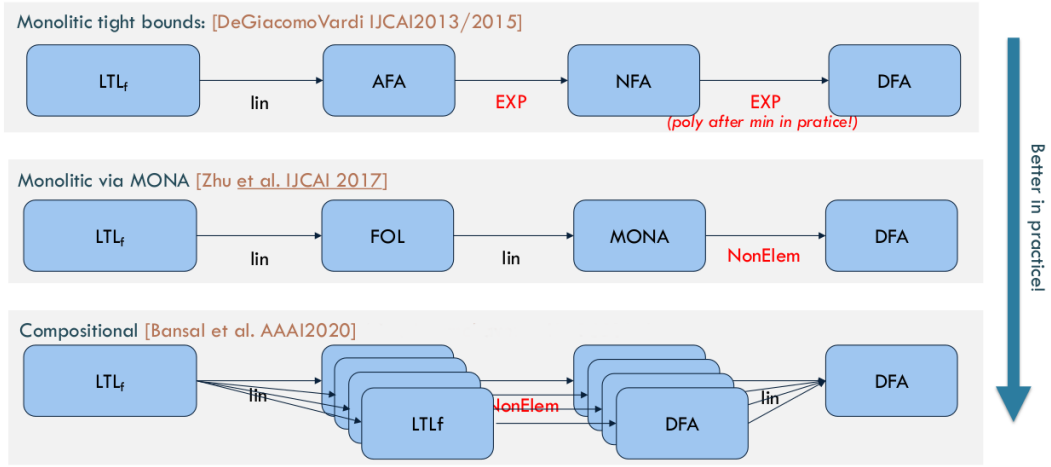


Figure 1.2. Overview of approaches to compute the DFA $\mathcal{A}_\varphi$ from an LTL$_f$ formula φ. At the top, the classical approach proposed in (De Giacomo and Vardi, 2013; De Giacomo and Vardi, 2015) reduces the problem to AFA determinization. In the middle, the translation is reduced to the translation from FOL-to-DFA (by using a state-of-the-art tool `Mona`) (Zhu, Tabajara, Li, et al., 2017), and a well-known correspondence between Weak-Monadic Second-order logic of one successor (WS1S) and automata (Büchi, 1960a; Elgot, 1961). At the bottom, we describe the approach proposed in (Bansal et al., 2020), where the outermost conjunction of the LTL$_f$ formula is decomposed into its operands, which are smaller LTL$_f$ formulas, and then each of these formulas is translated into DFA, again by resorting to FOL.

Giacomo, and Patrizi, 2018) for computing the temporal reward specification on-the-fly, and in (Xiao et al., 2021) to perform LTL$_f$ synthesis without fully constructing the DFA.

Finding good algorithms for the LTL$_f$/LDL$_f$-to-DFA transformation is of essential importance, as it is at the basis of solutions of many problems in AI. Part of this thesis will be dedicated to purse this goal.

1.2 Contributions

The contributions of the thesis can be split into the following parts:

- We introduce a novel, *compositional* approach, for the translation from an LTL$_f$/LDL$_f$ formula into a DFA. Moreover, we design a symbolic approach suitable for an effective implementation. Finally, we propose a tool, *Lydia*, that implements the compositional procedure and that on our benchmarks is highly competitive with the state-of-the-art tools for DFA construction and LTL$_f$ synthesis.

- We study different problems and develop new techniques for Reinforcement Learning with LTL$_f$/LDL$_f$ specifications. In particular, we investigate on the concept of "Restraining Bolts", where an authority imposes restraining LTL$_f$/LDL$_f$ specifications to the learning agent. Other contributions related to this part will be discussed below.

- We study a forward approach for LTL$_f$ synthesis, which drastically improves previous work on the topic (Xiao et al., 2021) thanks to a smarter way of

clustering automata transitions by means of knowledge compilation techniques. Moreover, we provide an implementation of such technique, showing its potential and its competitiveness with other state-of-the-art approaches, and break new ground for interesting and promising developments by further exploring the connection with FOND planning and forward search.

Figure Figure 1.3 visually depicts the areas of AI impacted by the thesis.

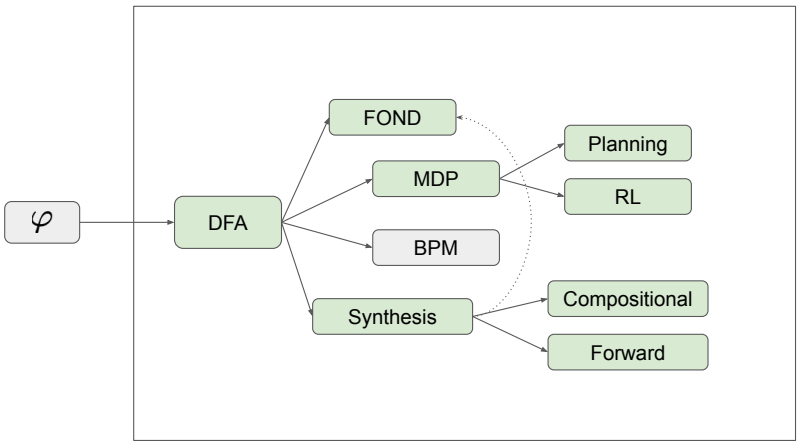


Figure 1.3. The diagram in Figure 1.1, expanded to include areas of AI in which the thesis has contributed to (highlighted in green). We consider *FOND planning* as area of contribution because it is a problem very similar to synthesis.

In the following, we briefly recap the main contributions in each category.

1.2.1 Compositional $\text{LTL}_f/\text{LDL}_f$-to-DFA

The first main contribution in this field is a novel solution to the $\text{LTL}_f/\text{LDL}_f$-to-DFA problem. In particular, we devise a new approach for the translation of LDL_f formulas into DFA by means of inductive translation rules that process each subformula separately, in a bottom-up fashion. That is why we call it *compositional*: the procedure tries to split the problem as much as possible, up to base cases (e.g. in LTL_f these are atomic formulas and boolean constants), and then, according to the LDL_f operator, the automata of the operands are composed using automata-theoretic operations (e.g. cartesian product, Kleene closure etc.). One of the main advantages of adopting this approach is practical: it allows to aggressively minimize partial results. This technique was already known to be effective by the authors of *Mona* (Klarlund, 1997; Klarlund and Møller, 2001) for the WS1S logic. Interestingly, the computational complexity of this translation approach is non-elementary (i.e. the time complexity cannot be bound by any tower of exponential in function of the input); nevertheless, as we will shall see, in practice the intractability does not emerge. Of course this scheme can be applied to LTL_f formulas as well, by first translating them into equivalent LDL_f formulas, which has a linear cost wrt the size of the formula.

The second contribution is to formalize the technique using semi-symbolic DFA representation. We say "semi-symbolic" because the transition function is represented symbolically (i.e. binary representation of the alphabet), whereas the state space is represented explicitly (i.e. unary representation). Symbolic representations, either in the alphabet of both in the state space and in the alphabet, are essential for the scalability of an implementation. We keep this contribution separate from the previous one as the proposed approach is agnostic with respect to the actual representation. and the semi-symbolic operations we used are non-trivial (in particular, the DFA-representation of an alternating finite automaton).

The third contribution of this part is an implementation of the abovementioned approach. The tool is called *Lydia*, and it is published open-source[1]. Benchmarks available from the literature on DFA constructions and LTL_f synthesis show that our implementation is competitive, and sometimes better, than other implementations. Our tool is able to perform synthesis by first constructing the entire (minimal) DFA of the $\text{LTL}_f/\text{LDL}_f$ formula, and then by relying on a symbolic technique used for LTL_f synthesis (Zhu, Tabajara, Li, et al., 2017). In particular, it is not just a scalable tool for DFA construction and synthesis of LTL_f formulas, but also the first able to handling LDL_f formulas.

1.2.2 Reinforcement Learning with $\text{LTL}_f/\text{LDL}_f$ reward specifications

The first contribution in this field is the study of a novel problem in AI, the *restraining bolts*. Restraining Bolts is a concept coming from the Science Fiction: a restraining bolt is a "device that restricts a droid's [agent's] actions when connected to its systems. Droid owners install restraining bolts to limit actions to a set of desired behaviors."[2]. In the context of AI, we imagined a scenario in which an authority wants to impose a restraining specification to the agent. Such entities, the authority and the agent, can be completely different, e.g. not even sharing the same perception of the world. Studying this problem from a classical Knowledge Representation perspective (Reiter, 2001) would require to establish some sort of "glue" between the representation by the agent and that by the restraining bolt. Instead, we bypass dealing with such a "glue" by studying this problem in the context of reinforcement learning, which is currently of great interest to develop components with forms of decision making, and in the case when the restraining specifications are expressed in temporal logic formalisms like $\text{LTL}_f/\text{LDL}_f$. We show formally, and illustrate with examples, that, under general circumstances, the agent can learn while shaping its goals to suitably conform (as much as possible) to the restraining bolt specifications.

Other contributions aim at further developing this concept, by studying variants of the problem, or addressing practical issues when training an agent in such settings.

The second contribution in this part is the study of the following problem. We have an *expert agent* which knows how to perform a certain task, and a (reinforcement) *learning agent*, which aims at learning the task of the expert, However, we assume that the expert is not able to "explain", or to directly transfer the knowledge to the learning agent, but only through demonstration of the optimal behaviour. We describe an Imitation Learning (IL) method where the execution traces generated by the expert agent are used to produce a logical specification of the reward function, to be incorporated into a restraining bolt. The restraining bolt can then be attached to the learning agent to drive the learning process and ultimately make it imitate

[1] https://github.com/whitemech/lydia

[2] https://www.starwars.com/databank/restraining-bolt

the expert. Crucially, we show that such method can be applied to *heterogeneous* agents, with the expert, the learner and the RB using different representations of the environment's actions and states, without specifying mappings among their representations.

The third contribution is to contribute to the techniques to handle temporal rewards and to the solutions to engineer them. In particular, following the work that studied MDPs with a set of $\text{LTL}_f/\text{LDL}_f$ specifications as source of reward (Brafman, De Giacomo, and Patrizi, 2018), we show how to compile the temporal specifications with a smaller state space overhead if we do not care which is the source of reward. Note that, when dealing with non-Markovian reward, it is necessary the overhead in order to keep track of relevant information from the history, and it is important to keep it as small as possible to make both planning and learning with MDPs as efficient as possible. We also introduce the novel concept of *monitoring rewards*, which drawn inspiration from the runtime monitoring literature (Bauer, Leucker, and Schallhart, 2010; Ly et al., 2013; De Giacomo, Masellis, Grasso, et al., 2014), allows a finer-grained reward specification at no additional overhead cost.

The fourth contribution is on the integration of planning and learning. The integration has many advantages in practical applications, as it allows for combining the different benefits of the two approaches: prediction of future states from planning with adaptivity to current situations from learning. However, a problem with is approach is that the two components should share a common representation of the information about the environment (e.g., states and actions). By exploiting the restraining bolts, we show how we can address the problem by using a modular design where the two components can use their own representation formalism, without requiring an explicit mapping between them. More specifically, we introduce the concept of domain-independent reward machines, generated by a goal-oriented planning system and use them to drive a reinforcement learning agent to reach a goal state. Moreover, we show how to automatically generate and use sub task decomposition to speed up the reinforcement learning process.

1.2.3 Forward Synthesis

The first contribution in this part is to devise a new method for solving LTL_f synthesis using a *forward* approach. Whilst other techniques relied on the full construction of the DFA, forward synthesis does not need to build the entire DFA but can build it on-the-fly, possibly exploring only a subset of the entire state space. This has the potential of being a very scalable technique, since many instances can be solved by searching for a winning strategy not far from the initial state of the agent-environment system. The method employs AND/OR graph search to find a winning strategy, with an implicit graph which is generated on-the-fly according to formula progression rules. Crucially, and this is what distinguishes our approach the most with the one proposed in (Xiao et al., 2021), we employ Knowledge Compilation (KC) techniques, in particular Sentential Decision Diagrams (SDD) (Darwiche, 2011), to cluster equivalent agent's and environment's moves in the AND/OR search graph, avoiding redundant checks on equivalent agent/env moves. Another useful feature of the chosen KC technique is the support for constant-time checks for state equivalence (only at syntactic level, not semantical).

The second contribution is to provide an implementation of the approach. It uses a recursive AND/OR search algorithm, able to handle cycles in the search graph. It uses state-of-the-art SDD compiler and implements procedures for LTL_f formula

progression. The tool is called *Cynthia*, and its source code is publicly available[3]. We experimentally showed that for a certain class of problems it outperforms state-of-the-art tools that rely on full construction of the DFA.

The deeper connection between LTL_f synthesis and AND/OR graph search highlighted in the contributions of this part opens countless developments, especially regarding the connections with FOND planning. In fact, observe that in both problems there is a game between two players, the agent and the environment, where for every sequence of *inputs* from the environments (i.e. the fluents of the environment in FOND and assignment of environment variables in synthesis) there is an agent's *response* (i.e. an action in FOND planning and an assignment of output variables in synthesis) such that the agent always wins the game. Therefore, we consider our contributions to be abscribed also to the FOND planning community.

1.3 Structure of the Thesis

The thesis is divided in four parts:

- in Part I, we give preliminary background knowledge on several topics of interest for the entire thesis: automata theory, temporal logics, and the automata-logic connection.
- in Part II, we present our contributions for the $\text{LTL}_f/\text{LDL}_f$-to-DFA problem, presenting our compositional approach, its semi-symbolic formalization, and the implementation details of the *Lydia* tool.
- in Part III, we present our contributions to the theory of MDPs and reinforcement learning with $\text{LTL}_f/\text{LDL}_f$ temporal reward specifications.
- in Part IV, we present our contributions to forward LTL_f synthesis with reduction to AND/OR graph search and by exploiting knowledge compilation techniques for achieving a more efficient search.

The rest of the thesis is organised as follows:

- Chapter 2 presents the topic of automata theory in the setting of finite words. It briefly surveys the main automata formalisms studied in the literature, the operations over them, and different representations of automata based on the explicit or symbolic representation of the state space or the action space.
- Chapter 3 introduces the reader, from a technical perspective, temporal logics on finite traces. We give the syntax and semantics of LTL_f and LDL_f, as well as state results from the literature about the complexity of reasoning.
- Chapter 4 explains more in detail how reasoning with $\text{LTL}_f/\text{LDL}_f$ formulas is performed by means of automata-theoretic techniques. We list the main approaches available from the literature to build the DFA from an $\text{LTL}_f/\text{LDL}_f$ formula, and we provide several examples describing how they working.
- Chapter 5 begins Part II, and introduces the first main contribution, by describing the compositional approach for translating $\text{LTL}_f/\text{LDL}_f$ formulas to DFA, explaining the details, proof of correctness, and analysis of computational complexity. We then give several examples and discuss potential future works.

[3] https://github.com/whitemech/cynthia/

- Chapter 6 gives a more concrete formalization of the compositional approach presented in Chapter 5, important for efficient and scalable implementations of the technique.
- Chapter 7 describes *Lydia*, our implementation of the compositional approach. We explain at high-level how it works, the data structures and algorithms used, and report its performance compared with state-of-the-art tools over benchmarks from the literature of LTL_f synthesis.
- Chapter 8 gives the background knowledge on the topics of classical Reinforcement Learning and Reinforcement Learning with non-Markovian rewards specified by temporal logic specifications.
- Chapter 9 begins Part III, and introduces the concept of Restraining Bolts, formalizes the problem, describes the solution, and provides use-cases of the approach.
- Chapter 10 presents the imitation learning problem with heterogeneous agents. We formalize the problem, propose a solution based on model learning techniques, and show use cases.
- Chapter 11 studies non-Markovian rewards specifications in $\text{LTL}_f/\text{LDL}_f$ in Markov Decision Processes with the aim of minimizing the state space overhead to handle the non-Markovianity. This is achieved by using reward transducers, i.e. transducers from fluents to reward signals. We also introduce monitoring reward specifications, a type of specification with finer-grained control of reward at no additional cost in terms of overhead wrt traditional specifications.
- Chapter 12 proposes an approach to integrate a planning module and a learning module by means of restraining bolts. We formalize the problem using the options framework and provide experimental coverage that shows the goodness of the approach.
- Chapter 13 begins Part IV of the thesis. It introduces the problem of LTL_f synthesis, as well as AND-OR graph search and Sentential Decision Diagrams, important building blocks for the forward LTL_f synthesis technique.
- Chapter 14 develops a forward LTL_f synthesis approach, based on AND-OR graph search, which efficiently explores the search graph by employing knowledge compilation techniques. We formalize all the components and prove the correctness.
- Chapter 15 presents the tool *Cynthia*, the implementation of the forward LTL_f synthesis technique studied in the previous chapter, and experimentally shows the performances on LTL_f synthesis benchmarks.
- Chapter 16 concludes the thesis, by summarizing the results and giving final remarks.

Part I

Temporal Logics and Automata Theory

Chapter 2

Finite Automata Theory

In this chapter, we revise the theory of finite automata, i.e. the part of automata theory that deals with machines with *finite states* and *constant bounded memory* or, equivalently, the types of automata that are as expressive as regular expressions. For a more thorough treatment, the reader might refer to the standard textbooks on the topic, e.g. (Hopcroft, Motwani, and Ullman, 2006).

The rest of the chapter is divided as follows:

- In Section 2.1, we introduce Deterministic Finite Automata (DFA), a type finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string.
- In Section 2.2, we present two generalizations of DFAs, namely Nondeterministic Finite Automata (NFA) and Universal Finite Automata (UFA), where in general there is more than one run for a given string (they differ in the acceptance condition). We also present the ϵ-NFA formalism, useful to formalize certain operations betwen automata.
- In Section 2.3, we introduce Alternating Finite Automata (AFA), a type of automaton that is also able to represent both existential choice (as an NFA) and universal choice (as an UFA). Such formalism generalizes all the automata above, achieving greater succincteness although without increase of expressiveness.
- In Section 2.4, we present Binary Decision Diagrams (BDD), data structures that are used to represent a Boolean function. It is an important building block for the implementation of automata.
- In Section 2.5, we survey the main categories of automata representation: fully explicit, semi-symbolic, and fully symbolic. For our purposes, from an implementation perspective, the semi-symbolic and fully-symbolic are the most relevant.
- In Section 2.6, we formalize some operations over DFAs that have a corresponding semantics with respect to the languages that they represent. In particular, we will see how we can compute the boolean operations, the concatenation, the Kleene closure and language projections of the operands languages exploiting their automata representation.
- Section 2.7 concludes the chapter.

2.1 Deterministic Finite Automata (DFA)

A *deterministic finite automaton* (DFA) (Rabin and Scott, 1959) $\mathcal{A}$ is a tuple $(Q, \Sigma, q_0, \delta, F)$ where Q is a finite non-empty set of states, Σ is a finite set of symbols called the *alphabet*, q_0 is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times \Sigma \to Q$ is the transition function.

A *run* of $\mathcal{A}$ on a *finite* word $w = a_0 \cdots a_n \in \Sigma^*$ is a finite sequence $q_1 \cdots q_n$ such that $q_{i+1} = \delta(q, a)$ for $0 \leq i < n$. The transition function δ can be extended into a function $\delta^* : Q \times \Sigma^* \to Q$ such that $\delta^*(q, \epsilon) = q$, and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$, for $q \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$. An automaton $\mathcal{A}$ *accepts* w if $\delta^*(q, w) \in F$. The *language of* $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts.

DFAs are closed under boolean operations. The *complement* of a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ is $\overline{\mathcal{A}} = (Q, \Sigma, q_0, \delta, S \setminus F)$, i.e. the complementation is obtained by inverting the acceptance of the automaton states. It can be shown that $\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})} = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. The *product under a boolean binary operator* $\odot \in \{\cup, \cap\}$ of $\mathcal{A}_1 = (Q_1, \Sigma, q_{10}, \delta_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, q_{20}, \delta_2, F_2)$, denoted as $\mathcal{A}_\odot$, is defined as $\mathcal{A}_\odot = (Q_1 \times Q_2, \Sigma, (q_{10}, q_{20}), \delta', F')$, where $\delta'((s_1, s_2), a) = (s_1', s_2')$ iff $s_1 = \delta_1(s_1, a)$ and $s_2 = \delta_2(s_2, a)$, and $F' = \{(q_1, q_2) \mid q_1 \in F_1 \odot q_2 \in F_2\}$. From a language-theoretic perspective, it can be showed that $\mathcal{L}(\mathcal{A}_\odot) = \mathcal{L}(\mathcal{A}_1) \odot_s \mathcal{L}(\mathcal{A}_2)$, where $\odot_s$ is the set operator analogous to the boolean operator $\odot$. The *intersection* and the *union* of two automata is equivalent to the product under disjunction and product under conjunction, respectively.

A DFA $\mathcal{A}$ is *minimal* if there is no other DFA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ and $|Q'| < |Q|$. It can be shown that if a DFA is minimal, it is unique for the language it represents (modulo state renaming). Or, equivalently, that each language $\mathcal{L} \subseteq \Sigma$ has one and only one minimal DFA that represents it. Any DFA can be minimized, and the operation is called *minimization*. The best known complexity for DFA minimization has complexity $\mathcal{O}(n \log n)$ (Hopcroft, 1971).

The DFAs are also closed under the following operations: existential and universal projection (followed by determinization), concatenation, closure and Kleene closure. More details on such operations will be provided later in this section.

2.2 Nondeterministic Finite Automata (NFA) and Universal Finite Automata (UFA)

A nondeterministic finite automaton (NFA) (Rabin and Scott, 1959) is defined as a DFA except for δ which be which becomes a relation rather than a function, i.e. $\delta : Q \times \Sigma \to 2^Q$. Therefore, a DFA can be seen as a special case of a NFA. Given a symbol σ, the successors of an NFA state q can be many, in contrast with the case of a DFA, in which only one successor was allowed. The notion of acceptance of a word w takes into account all the possible runs, and we say that an NFA *accepts* a word w if at least one of the runs hits an accepting state. The definition of the language of an NFA $\mathcal{A}_N$ is the same of the DFA, but using the new definition of acceptance.

Using the subset (or powerset) algorithm, an NFA $\mathcal{A}_N$ can be converted to an equivalent DFA $\mathcal{A}'$ that accepts the same language of $\mathcal{A}_N$, i.e. $\mathcal{L}(\mathcal{A}_N) = \mathcal{L}(\mathcal{A}')$ (Rabin and Scott, 1959). This operation is also called *determinization*. Let $\mathcal{A}_N = \langle Q, \Sigma, q_0, \delta, F \rangle$ be an NFA. The equivalent DFA $\mathcal{A}_d$ is defined as $\mathcal{A}_d = \langle 2^Q, \Sigma, \{q_0\}, \delta_d, F_d \rangle$, where $F_d = \{T \mid T \cap F \neq \emptyset\}$ is the collection of sets of states that intersect F nontrivially, and $\delta_d(T, a) = \{t \mid t \in \delta(s, a) \text{ for some } s \in T\}$.

In general, given an NFA $\mathcal{A}_N$ with a number of states n, the equivalent (minimal) DFA can have a number of states that is exponentially larger than n, i.e. 2^n. From

the above it follows that NFA and DFA are equally expressive formalisms, with NFA being, in general, exponentially more succinct than DFA.

We also define the *universal finite automaton* (UFA) to be structurally the same of an NFA, except for the acceptance condition: *every* run generated by a word w must be accepted in order to be accepted by the automaton.

2.2.1 NFA with ϵ-transitions: ϵ-NFA

An alternative definition of NFA, called ϵ-NFA includes a special symbol ϵ. Transitions labeled with ϵ can be taken without consuming any symbol of the input word. This new capability does not expand the class of languages that can be accepted by NFAs, but we introduce it as it gives us some "programming convenience", as we shall see.

2.3 Alternating Finite Automata (AFA)

An *alternating finite automaton* (AFA) (Chandra, Kozen, and Stockmeyer, 1981; Vardi, 1995) is defined as DFA and NFA, except for δ that is defined as $\delta : Q \times \Sigma \to B^+(Q)$, where $B^+(Q)$ is a set of positive boolean formulas whose atoms are states of Q. Due to the universal quantification, a run is represented by a run tree. An AFA $\mathcal{A}_A$ accepts a word w, if there exists a run tree on w such that every path ends in an accepting state. Let $\phi = \delta(q, \sigma)$ for some state q and some symbol σ. A *deterministic* transition is a transition in which ϕ is an atomic formula. An *existential transition* is a transition in which ϕ is made of only disjunctions. An NFA can be seen as a special case of an AFA in which the transitions are either deterministic or existential, whereas an UFA can be seen as a special case of an AFA in which all the transitions are either deterministic or universal.

Because of the universal choice in alternating transitions, a run of an alternating automaton is a tree rather than a sequence. A *tree* is a (finite or infinite) connected directed graph, with one node designated as the *root* and denoted by ε, and in which every non-root node has a unique parent (s is the *parent* of t and t is a *child* of s is there is an edge from s to t) and the root ε has no parent. The *level* of a node x, denoted $|x|$, is its distance from the root ε; in particular, $|\varepsilon| = 0$. A *branch* $\beta = x_0, x_1, \ldots$ of a tree is a maximal sequence of nodes such that x_0 is the root ε and x_i is the parent of x_{i+1} for all $i > 0$. Note that β can be finite or infinite. A Σ-*labeled tree*, for a finite alphabet Σ, is a pair $(\tau, \mathcal{T})$, where τ is a tree and $\mathcal{T}$ is a mapping from $nodes(\tau)$ to Σ that assigns to every node of τ a label in Σ. We often refer to $\mathcal{T}$ as the labeled tree. A branch $\beta = x_0, x_1, \ldots$ of $\mathcal{T}$ defined an infinite word $\mathcal{T}(\beta) = \mathcal{T}(x_0), \mathcal{T}(x_1), \ldots$ consiting of the sequence of labels along the branch.

Formally, a run of A on a finite word $w = a_0, a_1, \ldots, a_{n-1}$ is a finite Q-labeled tree r such that $r(\varepsilon) = q_0$ and the following holds: if $|x| = 1 < n$, $r(x) = q$, and $\delta_A(s, a_i) = \theta$, then x has k children $x_1, \ldots, x_k$ for some $k \leq |Q|$, and $\{r(x_1), \ldots, r(x_k)\}$ satisfies θ.

For example, if $\delta_A(q_0, a_0) = (q_1 \vee q_2) \wedge (q_3 \vee q_4)$, then the nodes of the run tree at level 1 include the label q_1 or the label q_2 and also include the label q_3 or the label q_4. Note that the depth of r (i.e., the maximal level of a node in r) is at most n, but not all branches need to reach such depth, since if $\delta_A(r(x), a_i) = \mathit{true}$, then x does not need to have any children. On the other hand, if $|x| = i < n$ and $r(x) = q$, then we cannot have $\delta_A(q, a_i) = \mathit{false}$, since false is not satisfiable. The run tree r is *accepting* if all nodes at depth n are labeled by states in F. Thus, a branch in an

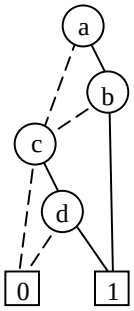


Figure 2.1. The BDD for the boolean function $F = ab + cd$.

accepting run has to hit the true transition or hit an accepting state after reading all the input word.

Given an AFA $\mathcal{A}_A$, we can compute an equivalent NFA $\mathcal{A}_N$ such that $\mathcal{L}(\mathcal{A}_A) = \mathcal{L}(\mathcal{A}_N)$ (Fellah, Jürgensen, and Yu, 1990). Let $\mathcal{A}_A = \langle Q, \Sigma, q_0, \delta, F \rangle$ be an AFA. The equivalent NFA $\mathcal{A}_n$ is defined as $\mathcal{A}_n = \langle 2^Q, \Sigma, \{q_0\}, \delta_n, F_n \rangle$, where $F_n = 2^F$ and $\delta_n(T, a) = \{T' \mid T' \text{ satisfies } \bigwedge_{t \in T} \delta(t, a)\}$.

This transformation may require an exponentially larger number of states in the resulting NFA $\mathcal{A}_N$, i.e. $2^{|Q|}$. Moreover, as shown in (Chandra, Kozen, and Stockmeyer, 1981), converting an n-state AFA to an equivalent DFA requires 2^{2^n} states in the worst case.

2.4 Binary Decision Diagrams (BDD)

In this section we present Binary Decision Diagrams (BDD), a well-studied knowledge compilation technique that it is a fundamental building block for non-naïve automata representations, discussed in Section 2.5.

A Binary Decision Diagram (BDD) (Bryant, 1992) is a data structure to represent Boolean functions. A BDD represents the function as a rooted directed acyclic graph. Each non-constant node n is labeled by a variable v and has edges directed towards two successor (children) nodes, $then(n)$ and $else(n)$, representing the cofactors of n with respect to v. The *then* successor is also called *high* and the *else* successor *low*. Each constant node is labeled with 0 or 1. For a given assignment of the variables, the value of the function is found by tracing a path from the root to a constant vertex following the branches indicated by the values assigned to the variables. The function value is given by the constant vertex label. For example, Figure 2.1 shows the BDD of the Boolean function $F = ab + cd$. The edges are directed downwards. The dashed edges (solid) edges correspond to $v = 0$ ($v = 1$).

In popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram (ROBDD in the literature, used when the ordering and reduction aspects need to be emphasized). The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order. This property makes it useful in functional equivalence checking and other operations like functional technology mapping.

In the average case, BDDs are a succinct representation for the set of models of a formula. Nevertheless, it is possible to prove that, for some orderings of the variables, the BDDs is exponential in the size of the formula. In addition, checking the optimal ordering (i.e., the ordering for which the BDDs is the smallest one) is NP-complete.

2.5 DFA Representations

In this section, we revise the most important types of representations for DFAs. We mostly follow the taxonomy made in (Zhu, Tabajara, Pu, et al., 2021).

2.5.1 *Fully-Explicit*: Explicit State, Explicit Alphabet

A DFA is in *fully-explicit* representation if the DFA is represented with an explicit graph. Explicit data structures store separately each of the configurations of Q and F, and the pairs of configurations of δ; typical examples are lists and hash tables. Their distinctive feature is that the memory needed to store a set is proportional to the number of its elements.

For domains in which either Q or Σ are very large, this solution is impractical. For example, in the context of automated software/hardware verification, e.g. model checking (Clarke, Grumberg, and Peled, 1999), the configurations of the systems are partitioned, or encoded, using a binary encoding with $\mathcal{P}$ boolean variables. Considering all the possible configurations, this yields an alphabet of size $2^{|\mathcal{P}|}$, i.e. exponential in the number of variables. In such cases, an explicit representation of the alphabet should be avoided.

2.5.2 *Semi-Symbolic*: Explicit State, Symbolic Alphabet

A DFA is in *semi-symbolic* representation if the state space Q is stored explicitly as in the automaton representation, as transitions are represented symbolically by propositional formulas but the states are still represented explicitly.

Such representation aims to address the issue of how to represent the transition function δ efficiently. It could be represented by a table mapping states and assignments in $2^{\mathcal{P}}$ to the set of successor states, but this table would necessarily be exponential in the number of propositions. Note that an alphabet of the form $2^{\mathcal{P}}$ is isomorphic to $\mathbb{B}^k$, where the vector $v \in \mathbb{B}^k$ identifies a subset $\Pi \subseteq \mathcal{P}$, such that the bit v_i is true iff $p_i \in \Pi$. Moreover, observe that any DFA over alphabet Σ can be reduced to a semi-symbolic representation, using a set of propositions $b_0, \ldots, b_n$ where $n = \lceil \log_2(|\Sigma|) \rceil$.

In practice, from a given state it is usually the case that multiple assignments can lead to a same successor state. These assignments can then be represented collectively by a single Boolean formula λ. For a given state, the number of such formulas is usually much smaller than the number of assignments. Therefore, the transition function can alternatively be represented by a relation $H : Q \times \Lambda \times Q$, where Λ is a set of propositional formulas over $\mathcal{P}$. We then have $(q_1, \lambda, q_2) \in H$ for a formula λ, iff $q_2 \in \delta(q, \sigma)$ for every $\sigma \in 2^{\mathcal{P}}$ that satisfies λ. Intuitively, the tuples of H can be thought of as edges in the graph representation of the automaton, labeled by the propositional formulas that match the transitions. It should be noted that MONA (Klarlund, 1997), explain in later chapters, adopts this representation, representing propositional formulas as Binary Decision Diagrams (BDDs) (Bryant, 1992).

Therefore, we call the above a semi-symbolic automaton representation, as transitions are represented symbolically by propositional formulas but the states are still represented explicitly. This may cause scalability problems when the state space is very large.

2.5.3 *Fully-Symbolic*: Symbolic State, Symbolic Alphabet

A DFA in *fully-symbolic* (*symbolic* for short) representation, if both states and transitions are represented symbolically. In the fully-symbolic representation, not only the symbols of the alphabet, but also states are encoded using a set of state variables $\mathcal{Z}$, where a state corresponds to an assignment of $\mathcal{Z}$.

Definition 2.1 (Symbolic Deterministic Finite Automaton)**.** *A* symbolic DFA *of a corresponding explicit* DFA $\mathcal{A} = \langle Q, 2^{\mathcal{P}}, q_0, \delta, F \rangle$*, in which* δ *is in the form of* $\delta : Q \times 2^{\mathcal{P}} \to Q$*, is represented as a tuple* $\mathcal{A}' = \langle \mathcal{Z}, \mathcal{P}, I, \delta', f \rangle$*, where*

- $\mathcal{Z}$ *is a set of state variables with* $|\mathcal{Z}| = \lceil \log_2 |Q| \rceil$*, and every state* q *in the explicit* DFA *corresponds to an assignment* $\mathcal{Z} \in 2^{\mathcal{Z}}$ *of propositions in* $\mathcal{Z}$*;*
- $\mathcal{P}$ *is the set of propositions as in* $\mathcal{A}$*;*
- $I \in 2^{\mathcal{Z}}$ *is the initial assignment corresponding to* q_0*;*
- $\boldsymbol{\delta} : 2^{\mathcal{Z}} \times 2^{\mathcal{P}} \to 2^{\mathcal{Z}}$ *is the transition function. Given assignment* Z *of current state* q *and transition condition* σ*,* $\boldsymbol{\delta}(Z, \sigma)$ *returns the assignment* Z' *corresponding to the successor state* $q' = \delta(q, \sigma)$*;*
- f *is a propositional formula over* $\mathcal{Z}$ *describing the accepting states, that is, each satisfying assignment* Z *of* f *corresponds to an accepting state* $q \in F$*.*

Since the states are encoded into a logarithmic number of state variables, depending on the structure of these formulas, the symbolic representation can be exponentially smaller than the semi-symbolic representation.

We distinguish two types of symbolic representations, *monolithic* and *partitioned* symbolic representations, which they differ only in how $\boldsymbol{\delta}$ is represented.

Monolithic

We call a symbolic representation *monolithic* if the representation of the transition function $\boldsymbol{\delta}$ is stored in a unique data structure.

Definition 2.2. *Let* $\mathcal{A}' = \langle \mathcal{Z}, \mathcal{P}, I, \boldsymbol{\delta}, f \rangle$ *be a symbolic* DFA*. The monolithic representation of* $\boldsymbol{\delta}$ *is a boolean formula* $\mathcal{T}$ *over variables* $\mathcal{Z} \cup \mathcal{P} \cup \mathcal{Z}'$*, where* $\mathcal{Z}' = z'_1, \ldots, z'_n$ *are the primed counterparts of* $\mathcal{Z}$ *and encode the next state. When representing the next state of the transition function, the same encoding is used for an interpretation* Z' *over* $\mathcal{Z}'$*. Moreover,* $\mathcal{T}$ *is such that it is satisfied by interpretations* $Z \in 2^{\mathcal{Z}}$ *,* $P \in 2^{\mathcal{P}}$ *and* $Z' \in 2^{\mathcal{Z}'}$ *iff* $\delta(q, P) = q'$*, where* q *and* q' *are the states corresponding to* Z *and* Z'*.*

In (Bansal et al., 2020), they use monolithic DFAs in order to perform symbolic LTL_f synthesis when the state space gets very large.

Partitioned

Note that the transition function δ can be represented by an indexed family consisting of a Boolean formula δ_i for each state variable $z_i \in Z$, which when evaluated over an assignment to $Z \cup P$ returns the next assignment to z_i. A DFA in symbolic representation is *partitioned* if the transition function is split into $|\mathcal{Z}|$ functions $\delta_i : 2^{\mathcal{Z}} \times 2^{\mathcal{P}} \to \{0, 1\}$. Each δ_i determines the value of the i-th bit in the next successor state q'. This allows more compositionality, which helps in keeping the transition function representation relatively small.

An example of how this representation has been applied to LTL$_f$ synthesis can be found in (Zhu, Tabajara, Li, et al., 2017), where they decompose a monolithic transition function into a sequence of BDDs $\mathcal{B} = \langle B_0, B_1, \ldots, B_{n-1} \rangle$, $n = \lceil \log_2 |Q| \rceil$, where each B_i, when evaluated on an interpretation $(Z \cup P)$, computes the i-th bit in the binary encoding of state $\delta(Z, P)$.

2.6 DFA operations: Projections, Concatenation, Closures

In this section we describe some operations over DFA that are at the foundations of the techniques introduced later in this thesis. The following definitions of concatenation and Kleene closure operations on semi-symbolic automata have been already introduced in (Yu, Bultan, et al., 2008), whereas the description of the (existential) projection on semi-symbolic automata can be found in (Klarlund and Møller, 2001).

2.6.1 Existential projection

Let $\mathcal{A}$ be a DFA, over the alphabet $\Sigma = \mathbb{B}^k$, and let $1 \leq i \leq k$. The *existential projection on the i-th bit of* $\mathcal{A}$ is another DFA $\mathcal{A}'$ obtained with the following procedure:

1. remove from $\mathcal{A}$ the i-th track from all the transition labels; this operation in general, yields an NFA $\mathcal{A}_N$;
2. determinize $\mathcal{A}_N$ via the subset construction to obtain the DFA $\mathcal{A}'$.

In language-theoretic terms,the *existential project operation* of a language $\mathcal{L}$ over the i-th track is another language $\mathcal{L}'$ defined as:

$$\mathcal{L}' = \{w \mid \exists w' \in \mathcal{L} : w \text{ is identical to } w' \text{ except for the } i\text{-th track}\}. \tag{2.1}$$

In our context, differently from the literature, we make the projection operation stronger in the sense that the resuling automaton will be defined over a new alphabet $\Sigma' = \mathbb{B}^{k-1}$ where the i-th track is completely removed, and not just ignored. In the following, we will denote with EPROJECT$(\mathcal{A}, i)$, or alternatively EPROJECT$_i(\mathcal{A})$, the existential projection of $\mathcal{A}$ on the i-th bit, and with EPROJECT$(\mathcal{A}, I)$, with $I \subset \{1, \ldots k\}$, the existential projection over a set of bits I.

2.6.2 Universal Projection

The *universal projection on the i-th bit of* $\mathcal{A}$ is the same of the existential version, but using an UFA instead of an NFA in the first step. Notice that the only thing that changes is the acceptance condition: the acceptances of the states inside the macro-state, during the subset construction, are interpreted as *disjunction* in the case of existential alternation, as in NFAs, but in *conjunction* in the case of universal alternation, as in UFAs.

In language-theoretic terms,the *universal project operation* of a language $\mathcal{L}$ over the i-th track is another language $\mathcal{L}'$ defined as:

$$\mathcal{L}' = \{w \mid \forall w' \in \mathcal{L} : w \text{ is identical to } w' \text{ except for the } i\text{-th track}\}. \tag{2.2}$$

We denote UPROJECT$(\mathcal{A}, i)$, or alternatively UPROJECT$_i(\mathcal{A})$, the universal projection of DFA $\mathcal{A}$ on the i-th bit, and with UPROJECT$(\mathcal{A}, I)$, with $I \subset \{1, \dots k\}$, the universal projection over a set of bits I.

2.6.3 Concatenation

The DFA $\mathcal{A}$ is a concatenation-DFA of the DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ if $\mathcal{L}(\mathcal{A}) = \{w_1 w_2 \mid w_1 \in \mathcal{L}(\mathcal{A}_1) \wedge w_2 \in \mathcal{L}(\mathcal{A}_2)\}$. Let $\mathcal{A}_1 = (Q_1, \Sigma, q_{10}, \delta_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, q_{20}, \delta_2, F_2)$, the concatenation $\mathcal{A} = A_1 \mid A_2$ can be constructed as follows. Without loss of generality, we assume $Q_1 \cap Q_2$ is empty. We first construct an intermediate automaton $\mathcal{A}' = (Q', q_{10}, \Sigma,', \delta', F')$ where:

- $Q' = Q_1 \cup Q_2$
- $\Sigma' = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta'(q, \alpha 0) = q',$ if $\delta_1(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta'(q, \alpha 0) = q',$ if $\delta_2(q, \alpha) = q'$
- $\forall q \in Q_1, \delta'(q, \alpha 1) = q',$ if $q \in F_1$ and $\exists q' \in Q_2, \delta_2(q_{20}, \alpha) = q'$
- $F' = F_1 \cup F_2,$ if $q_{20} \in F_2; F_2,$ otherwise.

Then, $\mathcal{A} =$ PROJECT$(\mathcal{A}', k+1)$. Since both $\mathcal{A}_1$ and $\mathcal{A}_2$ are DFA, the subset construction happens only when there exists $q \in F_1$ such that $\exists \alpha, q', q'', \alpha \in \Sigma, q' \in Q_1, q'' \in Q_2, \delta_1(q, \alpha) = q', \delta_2(q_{20}, \alpha) = q''$

Example 2.3. *Let $\mathcal{L}_1 = \{00^*\}$ and $\mathcal{L}_2 = \{0 + 1\}$ be two languages over the alphabet $\mathbb{B}^1$. The DFAs in semi-symbolic representation for $\mathcal{L}_1$ and $\mathcal{L}_2$ are showed in Figure 2.2.*

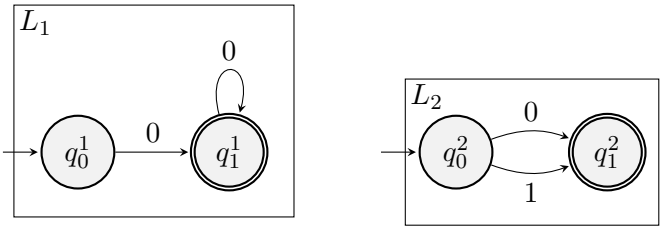


Figure 2.2. The automata for $\mathcal{L}_1$ (left) and $\mathcal{L}_2$ (right).

We are interested in computing the concatenation of the two languages, i.e. $\mathcal{L}_1 \mid \mathcal{L}_2$. To compute the corresponding automaton, we first add an auxiliary existential bit e, and set it to false (i.e. $\bar{e}$) to the existing transitions (Figure 2.3):

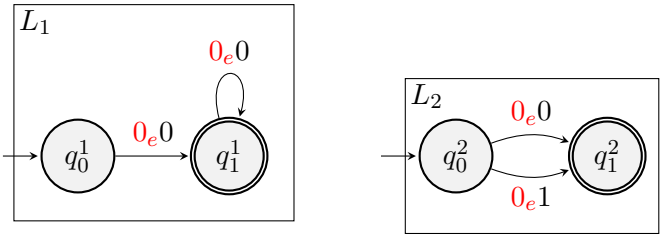


Figure 2.3. The automata $\mathcal{A}_1$ (left) and $\mathcal{A}_2$ with the new auxiliary existential bit.

Now, we add the concatenation transitions with bit e set to true (Figure 2.4):

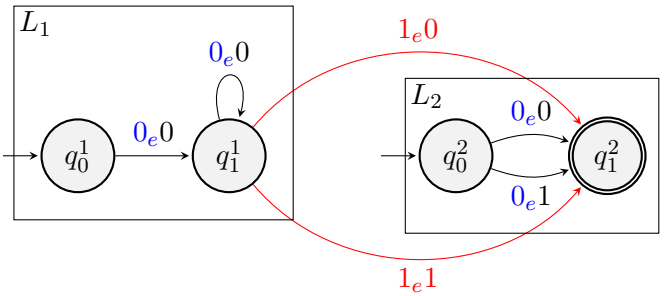


Figure 2.4. The automata $\mathcal{A}_1$ (left) and $\mathcal{A}_2$ with the new concatenating transitions.

We are ready to EPROJECT *the automaton on the new bit. In Figure 2.5 the concatenation automaton before projection, and in Figure 2.6 the result after the projection:*

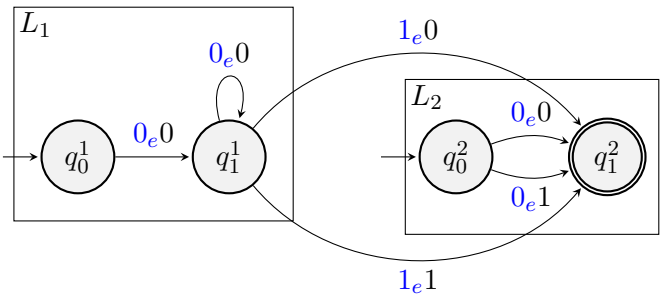


Figure 2.5. The concatenation automaton $\mathcal{A}_1 \mid \mathcal{A}_2$, before projection.

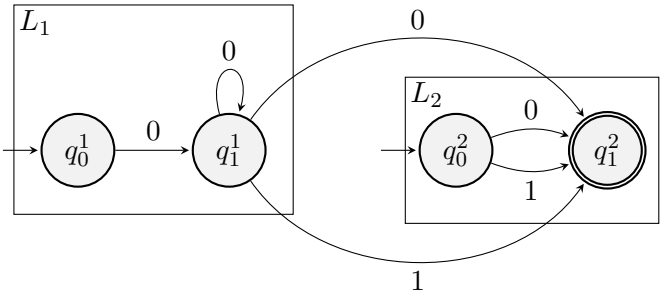


Figure 2.6. The concatenation automaton $\mathcal{A}_1 \mid \mathcal{A}_2$, after projection.

The next step is to determinize the output of EPROJECT*, since the operation introduced nondeterminism e.g. in* q_1^1 *(Figure 2.7):*

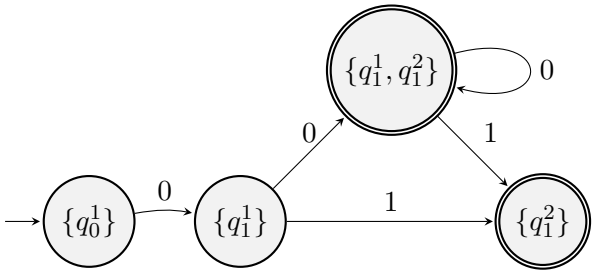


Figure 2.7. The concatenation automaton $\mathcal{A}_1 \mid \mathcal{A}_2$, after determinization.

After minimization (Figure 2.8):

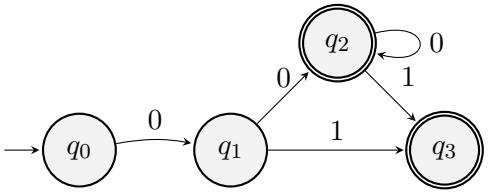


Figure 2.8. The concatenation automaton $\mathcal{A}_1 \mid \mathcal{A}_2$, after minimization.

A concatenation between two DFAs can be also done by means of ϵ-transitions, hence relying on the ϵ-NFA formalism. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ the two operands of the concatenation. To obtain the ϵ-NFA that represents $\mathcal{L}_1 \mid \mathcal{L}_2$:

- Connect the accepting states of $\mathcal{A}_1$ to the initial state of $\mathcal{A}_2$ by ϵ-transitions;
- Make all the states of $\mathcal{A}_1$ non-accepting.

Example 2.4. *Let consider the same languages and automata of Example 2.3. The concatenation can be obtained by adding ϵ-transitions from accepting states of $\mathcal{A}_1$ leading to the initial state q_0^2 of $\mathcal{A}_2$, and then by making q_1^1 non-accepting (Figure 2.9):*

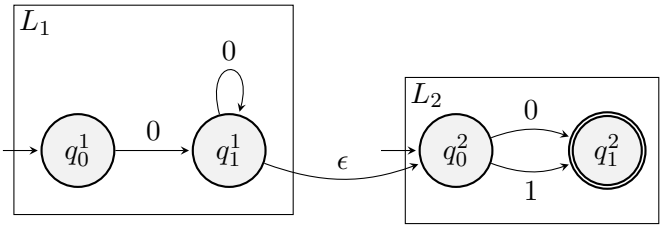


Figure 2.9. The concatenation automaton $\mathcal{A}_1 \mid \mathcal{A}_2$ using ϵ-transitions.

2.6.4 Kleene Closure

We first show how to compute a closure-DFA; from there, the construction of the Kleene Closure of a DFA is trivial. The DFA $\mathcal{A}$ is a closure-DFA of the DFA $\mathcal{A}_1$, if $\mathcal{L}(\mathcal{A}) = \{w_1 w_2 \dots w_k \mid k \geq 0 \wedge (\forall i.1 \leq i \leq k \implies w_i \in \mathcal{L}(\mathcal{A}_1))\}$ Let $\mathcal{A} = (Q_1, \Sigma, q_0, \delta, F)$ be an automaton. $\mathcal{L}(\mathcal{A})$ is a Kleene Closure of $\mathcal{A}_1$ if it also accepts the empty string, i.e. $L_k(\mathcal{A}) = L(A) \cup \{\epsilon\}$. Given $\mathcal{A}_1 = (Q_1, \Sigma, q_{10}, \delta_1, F_1)$, its closure $\mathcal{A}$ can be constructed by first constructing an intermediate DFA $\mathcal{A}' = (Q_1, \Sigma', q_{10}, \delta', F_1)$ as:

- $\Sigma' = \{\alpha 0 \mid \forall \alpha \in \Sigma\} \cup \{\alpha 1 \mid \forall \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta'(q, \alpha 0) = q',$ if $\delta_1(q, \alpha) = q'$.
- $\forall q \in Q_1, \delta'(q, \alpha 1) = q',$ if $q \in F_1$ and $\delta_1(q_{10}, \alpha) = q'$.

Then, $\mathcal{A} = \text{PROJECT}(\mathcal{A}', k+1)$ is the closure of $\mathcal{A}'$. Since $\mathcal{A}_1$ is a DFA, the project operation requires the subset construction only when there exists $q \in Q_1, q \in F_1$, and $\exists \alpha, q', q'', \alpha \in \Sigma, q', q'' \in Q_1, q' \neq q'', \delta_1(q_{10}, \alpha) = q', \delta_1(q_{10}, \alpha) = q''$. To have the Kleene closure, it would be enough to add the initial state to the set of accepting states.

Example 2.5. *Let $\mathcal{L} = \{0, 001\}$ be a language over the alphabet $\mathbb{B}^1$. The DFAs in semi-symbolic representation for $\mathcal{L}$ is showed in Figure 2.10:*

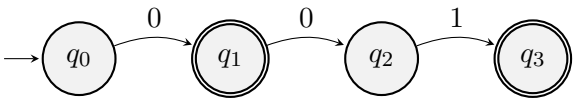


Figure 2.10. The automaton for $\mathcal{L}$.

We are interested in computing the Kleene closure of the language $\mathcal{L}$, $\mathcal{L}^*$. To compute the corresponding automaton, we first add an auxiliary existential bit e, and set it to false (i.e. $\bar{e}$) to the existing transitions (Figure 2.11):

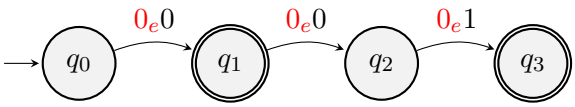


Figure 2.11. The automata $\mathcal{A}$ for $\mathcal{L}$ with the new auxiliary existential bit.

Now, we add the closure transitions, with the auxiliary bit set to 1 *(Figure 2.12):*

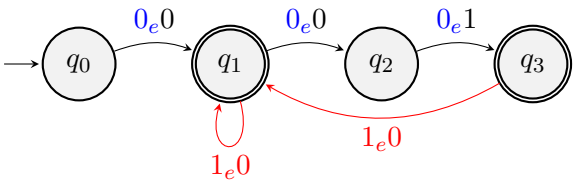


Figure 2.12. The automata $\mathcal{A}$ with the new concatenating transitions.

Now, we make the initial state accepting (Figure 2.13):

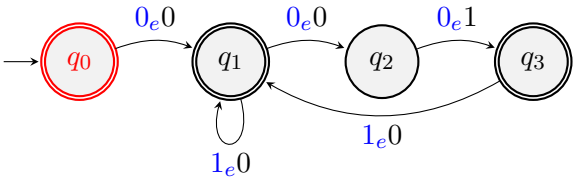


Figure 2.13. The automata $\mathcal{A}$ with the closure transitions and with the initial state marked as accepting.

We are ready to EPROJECT the automaton on the new bit. In Figure 2.14 the closure automaton before projection, and in Figure 2.15 the result after the projection:

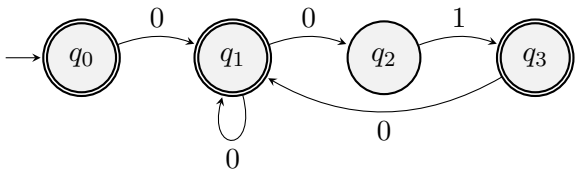


Figure 2.14. The Kleene closure automaton $\mathcal{A}^*$, before projection.

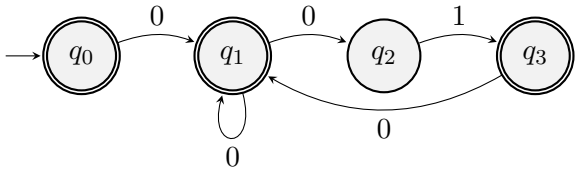


Figure 2.15. The Kleene closure automaton $\mathcal{A}^*$, after projection.

The next step is to determinize the output of EPROJECT, since the operation introduced nondeterminism e.g. in q_1 (Figure 2.16):

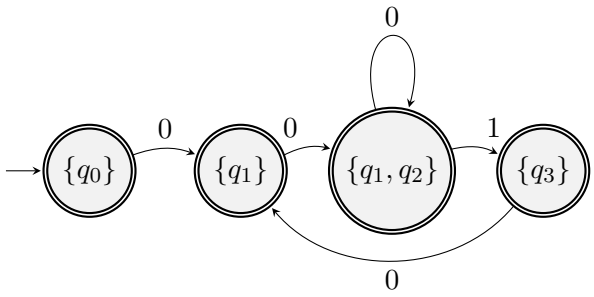


Figure 2.16. The concatenation automaton $\mathcal{A}^*$, after determinization.

After minimization (Figure 2.17):

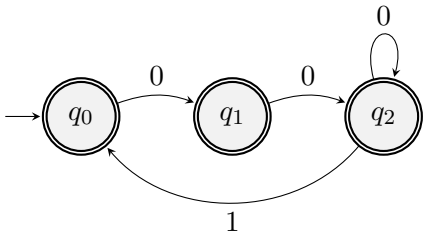


Figure 2.17. The Kleene closure automaton $\mathcal{A}^*$, after minimization.

The Kleene closure of a DFA can be constructed also by means of ϵ-transitions, hence relying on the ϵ-NFA formalism. Let $\mathcal{A}_1$ the operand of the Kleene closure operation. To obtain the ϵ-NFA that represents $\mathcal{L}_1^*$:

- Add a new initial state, make it accepting, and connect it to the old initial state of $\mathcal{A}_1$ by an ϵ-transition;
- Then, add ϵ-transitions from every accepting state of $\mathcal{A}_1$ to the old initial state.

Example 2.6. *Let consider the same language and automaton of Example 2.5. The concatenation can be obtained by adding a new initial accepting state q_i, connecting it with an ϵ-transition to q_0, and then by adding ϵ-transitions from q_1 and q_3 to q_i (Figure 2.18):*

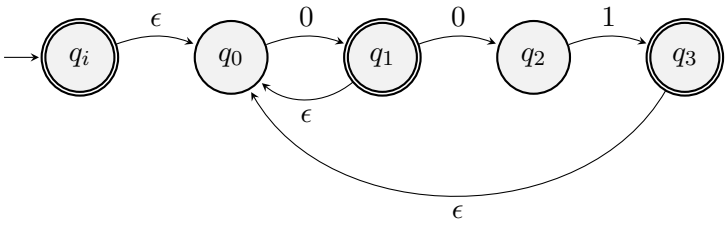


Figure 2.18. The concatenation automaton $\mathcal{A}_1 \mid \mathcal{A}_2$ using ϵ-transitions.

2.7 Summary

This chapter presented the main aspects of the theory of finite automata, which is one of the fundamental building block of the other topics of this thesis. We presented the formalisms of DFA, NFA, UFA and AFA, explained their relationship in terms of expressiveness and succinctness and mutual reducibility. We talked about the Binary Decision Diagram, which is very useful to achieve compact and canonical representation of boolean functions, and once the compilation is done, it allows to answer several queries (satisfiability, propositional equivalence, model counting etc.) in polytime. We discussed the main types of representations of DFAs, and relevant works in the literature that make use of them, pointing out pros and cons of each. Finally, we formalized and presented with thorough examples the main operations over DFAs in semi-symbolic representation, that will be relied on in future chapters.

Chapter 3

Temporal Logics on Finite Traces

In this chapter, we introduce the reader to the main important framework to talk about behaviours over time, which gives the foundations for our approach. The chapter is structured as follows:

- In Section 3.1, we talk about the well-known Linear-Time Temporal Logic (LTL) and its main applications.
- In Section 3.2, we present a specific formalism, namely *Linear Temporal Logic over Finite Traces* LTL_f, which shares the same syntax of LTL but is interpreted over finite traces.
- In Section 3.3 we describe a formal language that resembles regular expressions, called RE_f, still interpreted on finite words.
- In Section 3.4 we describe Linear Dynamic Logic (LDL_f), a formalism that merges the declarativeness and convenience of LTL_f with the expressive power of RE_f. Interestingly, such formalism has greater expressive power than LTL_f despite belonging to the same complexity class.
- Section 3.6 concludes the chapter.

3.1 Linear Temporal Logic

Linear Temporal Logic (LTL) (Pnueli, 1977) is a temporal logic, and it is the most popular and widely used temporal logic in computer science, especially in formal verification of software/hardware systems, in AI to reasoning about actions and planning, and in the area of Business Process Specification and Verification to specify processes declaratively.

It allows to express temporal patterns about some property p, like *liveness* (p *will eventually happen*), *safety* (p *will never happen*) and *fairness*, combinations of the previous patterns (*infinitely often* p *holds*, *eventually always* p *holds*).

3.1.1 Syntax

A LTL formula φ is defined over a set of propositional symbols $\mathcal{P}$ and are closed under the boolean connectives, the unary temporal operator $\bigcirc$(*next-time*) and the binary operator $\mathcal{U}$ (*until*):

$$\varphi \quad ::= \quad A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

With $A \in \mathcal{P}$.

Additional operators can be defined in terms of the ones above: as usual logical operators such as $\vee, \Rightarrow, \Leftrightarrow, \mathit{true}, \mathit{false}$ and temporal formulas like *eventually* as $\Diamond\varphi \doteq \mathit{true}\,\mathcal{U}\,\varphi$, *always* as $\Box\varphi \doteq \neg\Diamond\neg\varphi$ and *release* as $\varphi_1 \mathcal{R} \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$.

Example 3.1. *Several interesting temporal properties can be defined in* LTL*:*

- Liveness*: $\Diamond\varphi$, which means "condition expressed by φ* at some time *in the future will be satisfied", "sooner or later φ will hold" or "eventually φ will hold". E.g., $\Diamond rich$ (eventually I will become rich), Request $\Longrightarrow \Diamond Response$ (if someone requested the service, sooner or later he will receive a response).*
- Safety*: $\Box\varphi$, which means "condition expressed by φ,* every time *in the future will be satisfied", "always φ will hold". E.g., $\Box happy$ (I'm always happy), $\Box\neg(temperature > 30)$ (the temperature of the room must never be over 30).*
- Response*: $\Box\Diamond\varphi$ which means "at any instant of time there exists a moment later where φ holds". This temporal pattern is known in computer science as* fairness*.*
- Persistence*: $\Diamond\Box\varphi$, which stands for "There exists a moment in the future such that from then on φ always holds". E.g. $\Diamond\Box dead$ (at a certain point you will die, and you will be dead forever)*
- Strong fairness*: $\Box\Diamond\varphi_1 \Longrightarrow \Box\Diamond\varphi_2$, "if something is attempted/requested infinitely often, then it will be successful/allocated infinitely often". E.g., $\Box\Diamond ready \Longrightarrow \Box\Diamond run$ (if a process is in ready state infinitely often, then infinitely often it will be selected by the scheduler).*

3.1.2 Semantics

The semantics of LTL is provided by (infinite) *traces*, i.e. ω-word over the alphabet $2^{\mathcal{P}}$.

Definition 3.2. *Given a infinite trace π, we define that a* LTL *formula φ is* true *at time i, in symbols $\pi, i \models \varphi$ inductively as follows:*

$$\pi, i \models A, \text{ for } A \in \mathcal{P} \text{ iff } A \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$

$$\pi, i \models \bigcirc\varphi \text{ iff } \pi, i+1 \models \varphi$$

$$\pi, i \models \varphi_1 \mathcal{U} \varphi_2 \text{ iff } \exists j.(j \geq i) \wedge \pi, j \models \varphi \wedge \forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1$$

Similarly as in classical logic we give the following definitions:

Definition 3.3. *A* LTL *formula is* true *in* π*, in notation* $\pi \models \varphi$*, if* $\pi, 0 \models \varphi$*. A formula* φ *is* satisfiable *if it is true in some* π *and is* valid *if it is true in every* π*.* φ_1 entails φ_2*, in symbols* $\varphi_1 \models \varphi_2$ *iff* $\forall \pi, \forall i. \pi, i \models \varphi_1 \implies \pi, i \models \varphi_2$*.*

Now we state an important result:

Theorem 3.4 ((Sistla and Clarke, 1985))**.** *Satisfiability, validity, and entailment for* LTL *formulas are* PSPACE*-complete.*

Indeed, Linear Temporal Logic can be thought of as a specific decidable (PSPACE-complete) fragment of classical first-order logic (FOL).

3.2 Linear Temporal Logic on Finite Traces: LTL$_f$

Linear-time Temporal Logic over finite traces, LTL$_f$, is essentially standard LTL (Pnueli, 1977) interpreted over finite, instead of over infinite, traces (De Giacomo and Vardi, 2013). This apparently trivial difference has a big impact: as we will see, some LTL formula has a different meaning if interpreted over infinite traces or finite ones. The LTL$_f$ logic has been extensively used in Artificial Intelligence and Computer Science. For example, it is used in finite temporal synthesis (De Giacomo and Vardi, 2015; De Giacomo and Vardi, 2016; Camacho, Baier, et al., 2018; Zhu, Tabajara, Li, et al., 2017), in FOND planning with temporal specifications (Brafman and De Giacomo, 2019b; Camacho and McIlraith, 2019a), to express trajectory constraints in PDDL 3.0 (Bacchus and Kabanza, 1998; Gerevini et al., 2009b), in the theory of Markov Decision Processes to capture non-Markovian rewards (Bacchus, Boutilier, and Grove, 1996; Brafman, De Giacomo, and Patrizi, 2018; Brafman and De Giacomo, 2019a) with applications in reinforcement learning (Camacho, Icarte, et al., 2019; De Giacomo, Iocchi, et al., 2019; De Giacomo, Favorito, Iocchi, Patrizi, and Ronca, 2020), to specify business processes (Pešić, Bošnački, and Aalst, 2010), and many others.

3.2.1 Syntax

The syntax of LTL$_f$ is very similar of the one showed in Section 3.1.1. Given a set $\mathcal{P}$ of propositional symbols, LTL$_f$ formulae are built as follows:

$$\varphi \quad ::= \quad tt \mid \phi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

where tt is the tautology (not to be confused with $true = \phi \vee \neg\phi$), ϕ is a propositional formula over $\mathcal{P}$, $\bigcirc$ is the *next* operator, and $\mathcal{U}$ is the until operator.

We use the standard abbreviations for classical logic formulas:

$$\begin{aligned} \varphi_1 \vee \varphi_2 &\doteq \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \Rightarrow \varphi_2 &\doteq \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \Leftrightarrow \varphi_2 &\doteq \varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1 \\ true &\doteq \neg\phi \vee \phi \\ false &\doteq \neg\phi \wedge \phi \end{aligned}$$

And for temporal formulas:

$$\varphi_1 \mathcal{R} \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2) \tag{3.1}$$

$$\Diamond\varphi \doteq \mathit{true}\,\mathcal{U}\,\varphi \tag{3.2}$$

$$\Box\varphi \doteq \neg\Diamond\neg\varphi \tag{3.3}$$

$$\bullet\varphi \doteq \neg\bigcirc\neg\varphi \tag{3.4}$$

$$\mathit{Last} \doteq \bullet\mathit{false} \tag{3.5}$$

$$\mathit{End} \doteq \Box\mathit{false} \tag{3.6}$$

As the reader might already noticed, 3.2 and 3.3 are defined as in Section 3.1.1; Equation 3.1 is called *release*; Equation 3.4 is called *weak next* (notice that on finite traces $\neg\bigcirc\varphi \not\equiv \bigcirc\neg\varphi$); 3.5 denotes the end of the trace, while 3.6 denotes that the trace is ended.

Example 3.5. *Here we recall Example 3.1 and we see the impact on* Always, Eventually Response *and* Persistence LTL *formulas if interpreted on finite traces (i.e. formulas in* LTL$_f$*):*

- Safety*: $\Box A$ means that always* till the end of the trace φ *holds;*
- Liveness*: $\Diamond A$ means that eventually* before the end of the trace φ *holds;*
- Response*: $\Box\Diamond\varphi$ on finite traces becomes equivalent to* last point in the trace satisfies φ*, i.e. $\Diamond(\mathit{Last} \wedge \varphi)$. Intuitively, this is true because $\Box\Diamond\varphi$ implies that at the last point in the trace φ holds (because there are no successive instants of time that make φ true); but if this is the case, then what happens at previous points in the trace does not matter because the formula evaluates always to true, since as we just said φ must hold at the last point in the trace, hence the equivalence with $\Diamond(\mathit{Last} \wedge \varphi)$.*
- Persistence*: $\Diamond\Box\varphi$ on finite traces becomes equivalent to* last point in the trace satisfies φ*, i.e. $\Diamond(\mathit{Last} \wedge \varphi)$. Analogously to the previous case, the equivalence holds because $\Diamond\Box\varphi$ implies that at the last point in the trace $\Box\varphi$ holds (and so φ) since we have no further successive instants of time that make $\Box\varphi$ true. But if this is the case, then what happens at previous points in the trace does not matter because the formula evaluates always to true, since as we just said $\Box\varphi$ (and so φ) must hold at the last point in the trace, hence the equivalence with $\Diamond(\mathit{Last} \wedge \varphi)$.*

In other words, no direct nesting of eventually and always connectives is meaningful in LTL$_f$*, and this contrast what happens in* LTL *of infinite traces.*

Example 3.6. *Another remarkable evidence about the relevance of the assumption about the finiteness of traces is provided by the* DECLARE *approach (Pesic and Aalst, 2006).*

DECLARE *is a declarative approach to business process modeling based on* LTL *interpreted over finite traces. The intuition is to map finite traces describing a domain of interest (e.g. processes) into infinite traces under the assumption that*

$$\Diamond \mathit{end} \wedge \Box(\mathit{end} \Rightarrow \bigcirc \mathit{end}) \wedge \Box(\mathit{end} \Rightarrow \bigwedge_{p \in \mathcal{P}} \neg p) \tag{3.7}$$

which means that the following english statements hold:

- *end eventually holds ($end \notin \mathcal{P}$);*
- *once end is true, it is true forever;*
- *when end is true all other propositions must be false*

In other words, every finite trace π_f is extended with an infinite sequence of end, or in symbols $\pi_{inf} = \pi_f\{end\}^\omega$. By construction we have that

$$\pi_{inf} \models \Diamond end \wedge \Box(end \Rightarrow \bigcirc end) \wedge \Box(end \Rightarrow \bigwedge_{p \in \mathcal{P}} \neg p)$$

Despite it seems a nice construction to adapt LTL *on finite traces, in fact it is wrong due to the* next *operator: in an infinite trace a successor state always exists, whereas in a finite one this does not hold. There exists a counterexample showing that the interpretation of* LTL *formulas on finite traces with the construction just explained is* ***not*** *equivalent with proper interpretation over finite traces offered by* LTL$_f$*, i.e. in general:*

$$\pi_f\{end\}^\omega \models \varphi \not\Leftrightarrow \pi_f \models_f \varphi \tag{3.8}$$

To see why this is the case, consider the DECLARE *"negation chain succession" $\Box(a \Rightarrow \bigcirc\neg b)$ which requires that at any point in the trace, the state after we see a, b is false. Consider also the finite trace $\pi_f = \{a\}$ and the associated infinite trace $\pi_{inf} = \{a\}\{end\}^\omega$ built as explained before. We have that*

$$\pi_{inf} \models \Box(a \Rightarrow \bigcirc\neg b)$$

where $\models$ has been defined in 3.2. This is true because there is only one occurrence of a and then end holds forever (and so b does not).

But if the same formula is interpreted on finite traces (namely $\models_f$):

$$\pi_f \not\models_f \Box(a \Rightarrow \bigcirc\neg b)$$

because the finite trace a is true at the last instant, but then there is no next instance where b is false, so $\bigcirc\neg b$ is evaluated to false and the formula does not hold. The correct way to express "negation chain succession" on finite traces would be $\Box(a \Rightarrow \bullet\neg b)$.

The LTL *formulas φ that are insensitive to the problem just shown, i.e. such that*

$$\pi_f\{end\}^\omega \models \varphi \text{ iff } \pi_f \models_f \varphi \tag{3.9}$$

holds are defined insensitive to infiniteness *(De Giacomo, De Masellis, and Montali, 2014). This is another important evidence about the the relevance of the finiteness trace assumption.*

3.2.2 Semantics

The semantics of LTL$_f$ is given in terms of finite traces denoting a finite, possibly empty, sequence $\pi = \pi_0, \ldots, \pi_n$ of elements from the alphabet $2^{\mathcal{P}}$, containing all possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We denote the length of the trace π as $length(\pi) \doteq n + 1$, and with $Last(\pi) \doteq n$ the last index. We denote as $\pi(i) \doteq \pi_i$ the i-th step in the trace. If the trace is shorter and does not include an i-th step, $\pi(i)$ is undefined. We denote by $\pi(i, j) \doteq \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}$

the segment of the trace π starting at the i-th step and ending at the j-th step (excluded). If $j > length(\pi)$ then $\pi(i,j) = \pi(i, length(\pi))$. For every $j \leq i$, we have $\pi(i,j) = \epsilon$, i.e., the empty trace. Notice that, differently from (De Giacomo and Vardi, 2013), we allow the empty trace as in (Brafman, De Giacomo, and Patrizi, 2018) and (De Giacomo, Di Stasio, et al., 2020).

Definition 3.7. *Given a finite trace π, we inductively define when an LTL_f formula φ is satisfied at an instant $i \in \mathbb{N}$, in symbols $\pi, i \models \varphi$, as follows:*

$$\pi, i \models tt$$
$$\pi, i \models \phi \text{ iff } 0 \leq i < length(\pi) \text{ and } \pi(i) \models \phi$$
$$\pi, i \models \neg\varphi \text{ iff } \pi, i \models \varphi$$
$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$
$$\pi, i \models \bigcirc\varphi \text{ iff } 0 \leq i < length(\pi) - 1 \text{ and } \pi, i+1 \models \varphi \tag{3.10}$$
$$\pi, i \models \varphi_1 \,\mathcal{U}\, \varphi_2 \text{ iff for some } j \text{ s.t. } 0 \leq i \leq j < length(\pi), \text{ we have } \pi, j \models \varphi_2, \text{ and for all } k, i \leq k < j, \text{ we have } \pi, k \models \varphi_1 \tag{3.11}$$

Notice that Definition 3.7 is pretty similar to Definition 3.2, except the bounding of indexes in Equation 3.10 and Equation 3.11, to recognize that the trace is ended.

Analogously to Definition 3.3 we give the following definitions:

Definition 3.8. *A LTL_f formula is* true *in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is* satisfiable *if it is true in some π and is* valid *if it is true in every π. φ_1* entails *φ_2, in symbols $\varphi_1 \models \varphi_2$ iff $\forall\pi, \forall i.\pi, i \models \varphi_1 \implies \pi, i \models \varphi_2$.*

3.2.3 Complexity and Expressiveness

Thanks to reduction of LTL_f satisfiability (Definition 3.8) into LTL satisfiability for PSPACE membership and reduction of STRIPS planning into LTL_f satisfiability for PSPACE-hardness, as proposed in (De Giacomo and Vardi, 2013), we have this result:

Theorem 3.9 ((De Giacomo and Vardi, 2013))**.** *Satisfiability, validity and entailment for LTL_f formulas are* PSPACE*-complete.*

About expressiveness of LTL_f, we have that:

Theorem 3.10 ((De Giacomo and Vardi, 2013; Gabbay et al., 1997))**.** *LTL_f has exactly the same expressive power of* FOL *over finite ordered sequences.*

3.3 Regular Temporal Specifications (RE_f)

In this section, we talk about regular languages as a form of temporal specification over finite traces. In particular we focus on regular expressions (Hopcroft, Motwani, and Ullman, 2006).

A regular expression ϱ is defined inductively as follows, considering as alphabet the set of propositional interpretations $2^{\mathcal{P}}$, from a set of propositional symbols $\mathcal{P}$:

$$\varrho \quad ::= \quad \phi \mid \varrho_1 + \varrho_2 \mid \varrho_1; \varrho_2 \mid \varrho^*$$

where ϕ is a propositional formula that is an abbreviation for the union of all the propositional interpretations that satisfy ϕ, i.e. $\phi = \sum_{\Pi \models \phi} \Pi$ and $\Pi \in 2^{\mathcal{P}}$.

We denote by $\mathcal{L}(\varrho)$ the language recognized by a RE$_f$ expression. We interpret these expressions over finite traces, introduced in Section 3.2.2.

Definition 3.11. *We say that a regular expression* ϱ is true *in the finite trace* π *ifs* $\pi \in \mathcal{L}(\varrho)$. *We say that* ϱ is true at instant i *if* $\pi(i, last) \in \mathcal{L}(\varrho)$. *We say that* ϱ is true between instants i, j *if* $\pi(i, j) \in \mathcal{L}(\varrho)$.

Example 3.12. *We can express some of the formulas shown in Example 3.5, and many others, in* RE$_f$*:*

- Safety*:* φ^**, equivalent to* $\Box\varphi$
- Liveness*:* $true^*; \varphi; true^*$*, equivalent to* $\Diamond\varphi$*;*
- Response *and* Persistence*: as said before, when interpreted on finite traces, they are equivalent to* $\Diamond(Last \wedge \varphi)$*; hence, they can be rewritten in* RE$_f$ *as* $true^*; \varphi$
- Ordered occurrence*:* $true^*; \varphi_1; true^*; \varphi_2; true^*$*, equivalent to* $\Diamond(\varphi_1 \wedge \bigcirc\Diamond\varphi_2)$ *means that* φ_1 *and* φ_2 *happen in order;*
- Alternating sequence*:* $(\psi, \varphi)^*$ *means that* ψ *and* φ *alternate from the beginning of the trace, starting with* ψ *and ending with* φ*.*

The Alternating sequence *is an example of formula that has not a counterpart in* LTL$_f$*. More generally,* LTL$_f$ *(and* LTL*) are not able to capture regular structural properties on path (Wolper, 1981).*

This observation about expressiveness of RE$_f$ is confirmed by Theorem 6 of (De Giacomo and Vardi, 2013), which is a consequence of several classical results (Büchi, 1960b; Elgot, 1961; Trakhtenbrot, 1961; Thomas, 1979):

Theorem 3.13 ((De Giacomo and Vardi, 2013))**.** RE$_f$ *is strictly more expressive than* LTL$_f$

More precisely, RE$_f$ is expressive as *monadic second-order logic* MSO over bounded ordered sequences (Khoussainov and Nerode, 2001).

3.4 Linear Dynamic Logic on Finite Traces: LDL$_f$

The problem with RE$_f$ is that, although is strictly more expressive than LTL$_f$, is considered a low-level formalism for temporal specifications. For instance, RE$_f$ misses a direct construct for negation and for conjunction. Moreover, negation requires an exponential blow-up, hence adding complementation and intersection constructs are not advisable.

Linear Dynamic Logic of Finite Traces LDL$_f$ (De Giacomo and Vardi, 2013) merges LTL$_f$ with RE$_f$ in a very natural way, borrowing the syntax of PDL (Fischer and Ladner, 1979), a well-known (propositional) logic of programs in computer science. It keeps the declarativeness and convenience of LTL$_f$ while having the same expressive power of RE$_f$.

3.4.1 Syntax

Formally, LDL$_f$ formulas φ are built over a set of propositional symbols $\mathcal{P}$ as follows (Brafman, De Giacomo, and Patrizi, 2017):

$$\begin{array}{lcl} \varphi & ::= & tt \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\varrho\rangle\varphi \\ \varrho & ::= & \phi \mid \varphi? \mid \varrho_1 + \varrho_2 \mid \varrho_1; \varrho_2 \mid \varrho^* \end{array}$$

where tt stands for logical true; ϕ is a propositional formula over $\mathcal{P}$; ϱ denotes path expressions, which are RE over propositional formulas ϕ with the addition of the test construct $\varphi?$ typical of PDL. Moreover, we use the following abbreviations for classical logic operators:

$$\begin{aligned} \varphi_1 \vee \varphi_2 &\doteq \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \Rightarrow \varphi_2 &\doteq \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \Leftrightarrow \varphi_2 &\doteq \varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1 \\ ff &\doteq \neg tt \end{aligned}$$

And for temporal formulas:

$$[\varrho]\varphi \doteq \neg\langle\varrho\rangle\neg\varphi \tag{3.12}$$
$$End \doteq [true]ff \tag{3.13}$$
$$Last \doteq \langle true\rangle End \tag{3.14}$$

$[\varrho]\varphi$ and $\langle\varrho\rangle\varphi$ are analogous to box and diamond operators in PDL; Formula 3.14 denotes the last element of the trace, whereas Formula 3.13 denotes that the trace is ended. Intuitively, $\langle\varrho\rangle\varphi$ states that, from the current step in the trace, there exists an execution satisfying the RE ϱ such that its last step satisfies φ, while $[\varrho]\varphi$ states that, from the current step, all executions satisfying the RE ϱ are such that their last step satisfies φ.

3.4.2 Semantics

As we did in the previous sections, we formally give a semantics to LDL$_f$ (interpreted over finite traces, like LTL$_f$ and RE).

Definition 3.14. *Given a finite trace π, we define that a* LDL$_f$ *formula φ is* true *at time i ($0 \leq i \leq last$), in symbols $\pi, i \models \varphi$ inductively as follows:*

$\pi, i \models tt$
$\pi, i \models \neg\varphi$ *iff* $\pi, i \not\models \varphi$
$\pi, i \models \varphi_1 \wedge \varphi_2$ *iff* $\pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$
$\pi, i \models \langle\phi\rangle\varphi$ *iff* $i < last \wedge \pi(i) \models \phi \wedge \pi, i+1 \models \varphi$
$\pi, i \models \langle\psi?\rangle\varphi$ *iff* $\pi, i \models \psi \wedge \pi, i \models \varphi$
$\pi, i \models \langle\varrho_1 + \varrho_2\rangle\varphi$ *iff* $\pi, i \models \langle\varrho_1\rangle\varphi \vee \langle\varrho_2\rangle\varphi$
$\pi, i \models \langle\varrho_1; \varrho_2\rangle\varphi$ *iff* $\pi, i \models \langle\varrho_1\rangle\langle\varrho_2\rangle\varphi$
$\pi, i \models \langle\varrho^*\rangle\varphi$ *iff* $\pi, i \models \varphi \vee i < last \wedge \pi, i \models \langle\varrho\rangle\langle\varrho^*\rangle\varphi$ *and ϱ is not* test-only

We say that ϱ is test-only *if it is a* RE$_f$ *expression whose atoms are only tests, i.e. $\psi?$.*

Notice that LDL_f fully captures LTL_f. For every formula in LTL_f there exists a LDL_f formula with the same meaning, namely:

LTL_f	LDL_f
ϕ	$\langle\phi\rangle tt$
$\neg\varphi$	$\neg\varphi$
$\varphi_1 \wedge \varphi_2$	$\varphi_1 \wedge \varphi_2$
$\bigcirc\varphi$	$\langle true\rangle(\varphi \wedge \neg End)$
$\varphi_1 \mathcal{U} \varphi$	$\langle(\varphi_1?; true)^*\rangle(\varphi_2 \wedge \neg End)$

Notice also that every RE_f expression ϱ is captured in LDL_f by $\langle\varrho\rangle End$. Moreover, since also the converse holds, i.e. every LDL_f formula can be expressed in RE (by Theorem 11 in (De Giacomo and Vardi, 2013)), the following theorem holds:

Theorem 3.15 ((De Giacomo and Vardi, 2013))**.** *LDL_f has exactly the same expressive power of* MSO

Now we show several LDL_f examples.

Example 3.16. *Formulas described in Examples 3.5 and 3.12 can be rewritten in LDL_f as:*

- Safety*: $[true^*]\varphi$, equivalent to LTL_f formula $\Box\varphi$*
- Liveness*: $\langle true^*\rangle\varphi$, equivalent to LTL_f formula $\Diamond\varphi$*
- Conditional Response*: $[true^*](\varphi_1 \Rightarrow \langle true^*\rangle\varphi_2)$, equivalent to LTL_f formula $\Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$*
- Ordered occurrence*: $\langle true^*; \varphi_1; true^*; \varphi_2; true^*\rangle End$ equivalent to the RE_f expression $true^*; \varphi_1; true^*; \varphi_2; true^*$*
- Alternating occurrence*: $\langle(\psi; \varphi)^*\rangle End$ equivalent to the RE_f expression $(\psi; \varphi)^*$*

3.5 Reasoning in $\text{LTL}_f/\text{LDL}_f$

In this section, we study the complexity of $\text{LTL}_f/\text{LDL}_f$ reasoning (i.e. complexity of problems as defined in Definition 3.8, by leveraging the automata construction described in previous sections.

Theorem 3.17 ((De Giacomo and Vardi, 2013))**.** *Satisfiability, validity, and logical implication for LDL_f formulas are* PSPACE*-complete*

Proof. Given a $\text{LTL}_f/\text{LDL}_f$ φ, we can leverage Theorem 4.3 to solve these problems, namely:

- For $\text{LTL}_f/\text{LDL}_f$ satisfiability we compute the associated NFA (as explained in Chapter 4 (which is an exponential step) and then check NFA for nonemptiness (NLOGSPACE).

- For LTL$_f$/LDL$_f$ validity we compute the NFA associated to $\neg\varphi$ (which is an exponential step) and then check NFA for nonemptiness (NLOGSPACE).
- For LTL$_f$/LDL$_f$ logical implication $\psi \models \varphi$ we compute the NFA associated to $\psi \wedge \neg\varphi$ (which is an exponential step) and then check NFA for nonemptiness (NLOGSPACE).

□

3.6 Summary

In this chapter, we provided the logical tools to face other topics in later chapters. We introduced several formal languages, starting from the classical LTL. We then moved to logics interpreted on finite traces, namely LTL$_f$ and LDL$_f$, focusing on their interesting properties. These two formalisms will be extensively used in later chapters of this thesis.

Chapter 4

LTL_f and LDL_f translation to automata

Reasoning over $\text{LTL}_f/\text{LDL}_f$ is usually done by relying on automata theory. In particular, from a $\text{LTL}_f/\text{LDL}_f$ formula φ, we can build a deterministic finite automaton (DFA) $\mathcal{A}_\varphi$, whose alphabet is the set of propositional interpretations $\mathcal{P}$ of φ, that is semantically equivalent to the original formula (De Giacomo and Vardi, 2013; De Giacomo and Vardi, 2015). The size of $\mathcal{A}_\varphi$ can be double-exponentially larger than the original formula φ, in the worst case.

The rest of the chapter is structured as follows:

- in Section 4.1, we give the details of the aforementioned translation from $\text{LTL}_f/\text{LDL}_f$ formulas to alternating finite automata.
- in Section 4.2, we will see a more specialized algorithm that transforms an $\text{LTL}_f/\text{LDL}_f$ formula directly into an NFA, instead of passing through the AFA.
- in Section 4.3, we will go further and present a direct translation from $\text{LTL}_f/\text{LDL}_f$ formulas into DFA, hence bypassing the determinization step from NFA into DFA, but doing it on-the-fly.
- Section 4.4 presents another approach known in the literature, which first transforms an LTL_f formula into a FOL formula, and then applies the inductive translation rules defined for a kind of second-order logic, namely Weak-Monadic Second-Order Logic of one successor (WS1S).
- Section 4.5 concludes the chapter.

4.1 From $\text{LTL}_f/\text{LDL}_f$ to AFA

In this section, we describe more in detail the transformation from $\text{LTL}_f/\text{LDL}_f$ formulas to AFA.

Given an $\text{LTL}_f/\text{LDL}_f$ formula φ, an equivalent AFA $\mathcal{A}_\varphi = \langle Q, \Sigma, q_0, \delta, F \rangle$ can be constructed as follows:

- $\Sigma = 2^{\mathcal{P}}$;
- $Q = \mathsf{cl}(\varphi)$;
- $q_0 = \varphi$;

- $\delta(\psi, \Pi)$ defined in Section 4.1.1 for LTL$_f$ and in Section 4.1.2 for LDL$_f$;
- $F = \{\psi \mid \psi \in Q \text{ and } \delta(\psi, \epsilon)\}$

Where $\mathsf{cl}(\varphi)$ is the set of subformulas of φ. It can be shown that, by construction, $\pi \models \varphi$ iff $\pi \in \mathcal{A}_\varphi$.

Then, the DFA for φ can be obtained by determinizing $\mathcal{A}$.

4.1.1 ∂ function for LTL$_f$

The We give the following definition:

Definition 4.1. *The* delta function ∂ for LTL$_f$ formulas *is a function that takes as input an (implicitly quoted)* LTL$_f$ *formula φ in NNF and a propositional interpretation Π for $\mathcal{P}$, and returns a positive boolean formula whose atoms are (implicitly quoted) φ subformulas. It is defined as follows:*

$$
\begin{aligned}
\partial(tt, \Pi) &= true \\
\partial(ff, \Pi) &= false \\
\partial(A, \Pi) &= \begin{cases} true & \text{if } A \in \Pi \\ false & \text{if } A \notin \Pi \end{cases} \\
\partial(\neg A, \Pi) &= \begin{cases} false & \text{if } A \in \Pi \\ true & \text{if } A \notin \Pi \end{cases} \\
\partial(\varphi_1 \wedge \varphi_2, \Pi) &= \partial(\varphi_1, \Pi) \wedge \partial(\varphi_2, \Pi) \\
\partial(\varphi_1 \vee \varphi_2, \Pi) &= \partial(\varphi_1, \Pi) \vee \partial(\varphi_2, \Pi) \\
\partial(\mathsf{O}\varphi, \Pi) &= \varphi \wedge \neg End \equiv \varphi \wedge \Diamond true \\
\partial(\bullet\varphi, \Pi) &= \varphi \vee End \equiv \varphi \vee \Box false \\
\partial(\varphi_1 \,\mathcal{U}\, \varphi_2, \Pi) &= \partial(\varphi_2, \Pi) \vee (\partial(\varphi_1, \Pi) \wedge \partial(\mathsf{O}(\varphi_1 \,\mathcal{U}\, \varphi_2), \Pi)) \\
\partial(\varphi_1 \mathcal{R} \varphi_2, \Pi) &= \partial(\varphi_2, \Pi) \wedge (\partial(\varphi_1, \Pi) \vee \partial(\bullet(\varphi_1 \,\mathcal{R}\, \varphi_2), \Pi))
\end{aligned}
\tag{4.1}
$$

where End is defined as Equation 3.6. As a consequence of Definition 4.1 and from Equation 3.2 and 3.3, we can deduce that

$$
\begin{aligned}
\partial(\Diamond\varphi, \Pi) &= \partial(\varphi, \Pi) \vee \partial(\mathsf{O}\Diamond\varphi, \Pi) \\
\partial(\Box\varphi, \Pi) &= \partial(\varphi, \Pi) \wedge \partial(\bullet\Box\varphi, \Pi)
\end{aligned}
$$

Moreover, we define $\partial(\varphi, \epsilon)$ which is inductively defined as Equation 4.1, except

for the following cases:

$$\begin{aligned}
\partial(tt, \epsilon) &= \mathit{true} \\
\partial(\mathit{ff}, \epsilon) &= \mathit{false} \\
\partial(A, \epsilon) &= \mathit{false} \\
\partial(\neg A, \epsilon) &= \mathit{false} \\
\partial(\bigcirc\varphi, \epsilon) &= \mathit{false} \\
\partial(\bullet\varphi, \epsilon) &= \mathit{true} \\
\partial(\varphi_1 \,\mathcal{U}\, \varphi_2, \epsilon) &= \mathit{false} \\
\partial(\varphi_1 \,\mathcal{R}\, \varphi_2, \epsilon) &= \mathit{true}
\end{aligned} \tag{4.2}$$

Note that $\partial(\varphi, \epsilon)$ is always either *true* or *false*. It is worth to observe for future use that from Equation 4.2 we can say $\partial(\Diamond\varphi, \epsilon) = \mathit{false}$ and $\partial(\Box\varphi, \epsilon) = \mathit{true}$.

4.1.2 ∂ function for LDL$_f$

We give the following definition:

Definition 4.2. *The* delta function ∂ for LDL$_f$ formulas *is a function that takes as input an (implicitly quoted)* LDL$_f$ *formula* φ *in NNF, extended with auxiliary constructs* $\mathtt{F}_\psi$ *and* $\mathtt{T}_\psi$*, and a propositional interpretation* Π *for* $\mathcal{P}$*, and returns a positive boolean formula whose atoms are (implicitly quoted)* φ *subformulas (not*

including $\boldsymbol{F}_\psi$ *or* $\boldsymbol{T}_\psi$*). It is defined as follows:*

$$
\begin{array}{rcl}
\partial(tt,\Pi) & = & \mathit{true} \\
\partial(\mathit{ff},\Pi) & = & \mathit{false} \\
\partial(\phi,\Pi) & = & \partial(\langle\phi\rangle tt,\Pi) \\
\partial(\varphi_1\wedge\varphi_2,\Pi) & = & \partial(\varphi_1,\Pi)\wedge\partial(\varphi_2,\Pi) \\
\partial(\varphi_1\vee\varphi_2,\Pi) & = & \partial(\varphi_1,\Pi)\vee\partial(\varphi_2,\Pi) \\
\partial(\langle\phi\rangle\varphi,\Pi) & = & \begin{cases} \boldsymbol{E}(\varphi) & \text{if } \Pi\models\phi \\ \mathit{false} & \text{if } \Pi\not\models\phi \end{cases} \\
\partial(\langle\varrho?\rangle\varphi,\Pi) & = & \partial(\varrho,\Pi)\wedge\partial(\varphi,\Pi) \\
\partial(\langle\varrho_1+\varrho_2\rangle\varphi,\Pi) & = & \partial(\langle\varrho_1\rangle\varphi,\Pi)\vee\partial(\langle\varrho_2\rangle\varphi,\Pi) \\
\partial(\langle\varrho_1;\varrho_2\rangle\varphi,\Pi) & = & \partial(\langle\varrho_1\rangle\langle\varrho_2\rangle\varphi,\Pi) \\
\partial(\langle\varrho^*\rangle\varphi,\Pi) & = & \partial(\varphi,\Pi)\vee\partial(\langle\varrho\rangle\boldsymbol{F}_{\langle\varrho^*\rangle\varphi},\Pi) \\
\partial([\phi]\varphi,\Pi) & = & \begin{cases} \boldsymbol{E}(\varphi) & \text{if } \Pi\models\phi \\ \mathit{true} & \text{if } \Pi\not\models\phi \end{cases} \\
\partial([\varrho?]\varphi,\Pi) & = & \partial(\mathit{nnf}(\neg\varrho),\Pi)\vee\partial(\varphi,\Pi) \\
\partial([\varrho_1+\varrho_2]\varphi,\Pi) & = & \partial([\varrho_1]\varphi,\Pi)\wedge\partial([\varrho_2]\varphi,\Pi) \\
\partial([\varrho_1;\varrho_2]\varphi,\Pi) & = & \partial([\varrho_1][\varrho_2]\varphi,\Pi) \\
\partial([\varrho^*]\varphi,\Pi) & = & \partial(\varphi,\Pi)\wedge\partial([\varrho]\boldsymbol{T}_{\langle\varrho^*\rangle\varphi},\Pi) \\
\partial(\boldsymbol{T}_\psi,\Pi) & = & \mathit{true} \\
\partial(\boldsymbol{F}_\psi,\Pi) & = & \mathit{false}
\end{array}
\tag{4.3}
$$

where $\boldsymbol{E}(\varphi)$ *recursively replaces in* φ *all occurrences of atoms of the form* $\boldsymbol{T}_\psi$ *and* $\boldsymbol{F}_\psi$ *by* $\boldsymbol{E}(\psi)$*.*

Moreover, we define $\partial(\varphi,\epsilon)$ *which is inductively defined as Equation 4.3, except for the following cases:*

$$
\begin{array}{rcl}
\partial(\langle\phi\rangle\varphi,\epsilon) & = & \mathit{false} \\
\partial([\phi]\varphi,\epsilon) & = & \mathit{true}
\end{array}
\tag{4.4}
$$

Note that $\partial(\varphi,\epsilon)$ is always either *true* or *false*.

4.2 The LDL$_f$2NFA algorithm

Algorithm 1 (LDL$_f$2NFA) takes in input a LDL$_f$/LTL$_f$ formula φ and outputs a NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F\rangle$ that accepts exactly the traces satisfying φ. It is a variant of the algorithm presented in (De Giacomo and Vardi, 2015), and its correctness relies on the fact that every LDL$_f$/LTL$_f$ formula φ can be associated a polynomial *alternating automaton on words* (AFW) accepting exactly the traces that satisfy φ and that every AFW can be transformed into an NFA (De Giacomo and Vardi, 2013). The proposed algorithm requires that φ is in *negation normal form* (NNF), i.e. with negation symbols occurring only in front of propositions.

The function ∂ used in lines 5, 11 and 14 is the one defined in sections 4.1.1 and 4.1.2; whether we are translating a LTL$_f$ or a LDL$_f$ formula, we use the function ∂ from Definition 4.1 and from Definition 4.2, respectively.

Algorithm 1 LDL$_f$2NFA: from LTL$_f$/LDL$_f$ formula φ to NFA $\mathcal{A}_\varphi$

input LDL$_f$/LTL$_f$ formula φ
output NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F \rangle$
$q_0 \leftarrow \{\varphi\}$
$F \leftarrow \{\emptyset\}$
if $(\partial(\varphi, \epsilon) = \mathit{true})$ **then**
    $F \leftarrow F \cup \{q_0\}$
$Q \leftarrow \{q_0, \emptyset\}$
$\delta \leftarrow \emptyset$
while (Q or δ change) **do**
    for $(q \in Q)$ **do**
        if $(q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi))$ **then**
            $Q \leftarrow Q \cup \{q'\}$
            $\delta \leftarrow \delta \cup \{(q, \Pi, q')\}$
            if $(\bigwedge_{(\psi \in q')} \partial(\psi, \epsilon) = \mathit{true})$ **then**
                $F \leftarrow F \cup \{q'\}$

How LDL$_f$2NFA works

The NFA $\mathcal{A}_\varphi$ for a LDL$_f$ formula φ is built in a forward fashion. Until convergence is reached (i.e. states and transitions do not change), the algorithm visits every state q seen until now, checks for all the possible transitions from that state and collects the results, determining the next state q', the new transition (q, Π, q') and if q' is a final state. Intuitively, the delta function ∂ emulates the semantic behaviour of every LTL$_f$/LDL$_f$ subformula after seeing Π.

States of $\mathcal{A}_\varphi$ are sets of atoms (each atom is a quoted φ subformula) to be interpreted as conjunctions. The empty conjunction $\emptyset$ stands for *true*. q' is a set of quoted subformulas of φ denoting a minimal interpretation such that $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$ (notice that we trivially have $(\emptyset, p, \emptyset) \in \delta$ for every $p \in 2^{\mathcal{P}}$).

The following result holds:

Theorem 4.3 ((De Giacomo and Vardi, 2015))**.** *Algorithm* LDL$_f$2NFA *is correct, i.e., for every finite trace* π : $\pi \models \varphi$ *iff* $\pi \in \mathcal{L}(\mathcal{A}_\varphi)$*. Moreover, it terminates in at most an exponential number of steps, and generates a set of states* S *whose size is at most exponential in the size of the formula* φ*.*

In order to obtain a DFA, the NFA $\mathcal{A}_\varphi$ can be determinized in exponential time (Rabin and Scott, 1959). Thus, we can transform a LTL$_f$/LDL$_f$ formula into a DFA of double exponential size.

Example 4.4. *In this example we see a run of the Algorithm 1 with the* LTL$_f$ *formula* $\Box A$ *(*A *atomic).*

0. Set up:

$$
\begin{aligned}
q_0 &= \{\Box A\} \\
Q &= \{q_0, \emptyset\} \\
F &= \{q_0, \emptyset\} \qquad \textit{(because } \partial(\Box A, \epsilon) = \partial(\mathit{false}\, \mathcal{R}\, \neg A, \epsilon) = \mathit{true}\textit{)} \\
\delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset)\}
\end{aligned}
$$

1. *Iteration: analyze* $q = \{\Box A\}$

   - *with* $\Pi = \{A\}$ *we have*

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Box A, \Pi) \\ &\models \partial(A, \Pi) \wedge \partial(\bullet \Box A, \Pi) \\ &\models \mathit{true} \wedge (\text{“}\Box A\text{”} \vee \text{“}\Box \mathit{false}\text{”}) \end{aligned}$$

   Notice that $\mathit{true} \wedge (\text{“}\Box A\text{”} \vee \text{“}\Box \mathit{false}\text{”})$ *is a* propositional formula *with* LTL$_f$ *formulas as atoms. As a minimal interpretation we have both* $q' = \{\text{“}\Box A\text{”}\}$ *and* $q' = \{\text{“}\Box \mathit{false}\text{”}\}$*. Since in both cases we have that* $\partial(\psi, \epsilon) = \mathit{true}$*, at the end of the iteration we have:*

$$\begin{aligned} q_0 &= \{\Box A\} \\ Q &= \{q_0, \{\Box \mathit{false}\}, \emptyset\} \\ F &= \{q_0, \{\Box \mathit{false}\}, \emptyset\} \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset), \\ &\quad (q_0, \{A\}, q_0), (q_0, \{A\}, \{\Box \mathit{false}\})\} \end{aligned}$$

   - *with* $\Pi = \{\}$ *we have*

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Box A, \Pi) \\ &\models \partial(A, \Pi) \wedge \partial(\bullet \Box A, \Pi) \\ &\models \mathit{false} \wedge (\text{“}\Box A\text{”} \vee \text{“}\Box \mathit{false}\text{”}) \end{aligned}$$

   Which is always false. Thus we do not change nothing.

2. *Iteration: we already analyzed* $q = \{\Box A\}$*, so we analyze* $q = \{\Box \mathit{false}\}$

   - *Both with* $\Pi = \{\}$ *and* $\Pi = \{A\}$ *we have that:*

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Box false, \Pi) \\ &\models \partial(false, \Pi) \wedge \partial(\bullet \Box false, \Pi) \\ &\models \mathit{false} \wedge (\text{“}\Box false\text{”} \vee \text{“}\Box \mathit{false}\text{”}) \end{aligned}$$

   Which is always false. Thus we do not change nothing.

The NFA $\mathcal{A}_\varphi = \langle 2^{\{A\}}, Q, q_0, \delta, F\rangle$ *is depicted in Figure 4.1, whereas the associated* DFA *is in Figure 4.2.*

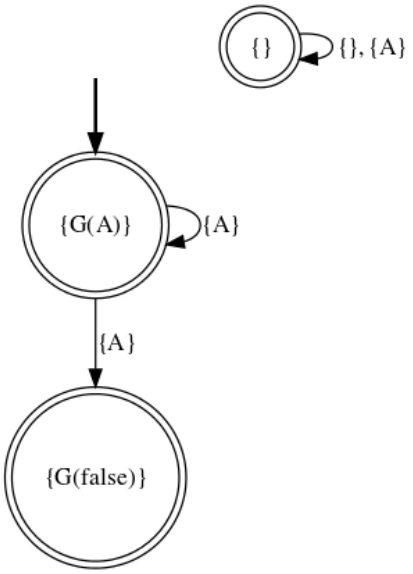


Figure 4.1. The NFA associated to $\Box A$. $G(A)$ stands for $\Box A$

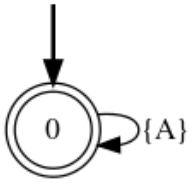


Figure 4.2. The DFA associated to $\Box A$

Example 4.5. *Analogously to what we did in 4.4, we see a run of the Algorithm 1, with the* LTL$_f$ *formula* $\Diamond A$ *(A atomic).*

0. Set up:

$$
\begin{aligned}
q_0 &= \{\Diamond A\} \\
Q &= \{q_0, \emptyset\} \\
F &= \{\emptyset\} \qquad \textit{(because } \partial(\Diamond A, \epsilon) = \partial(\textit{true}\,\mathcal{U}\, A, \epsilon) = \textit{false)} \\
\delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset)\}
\end{aligned}
$$

1. Iteration: analyze $q = \{\Diamond A\}$

- *with* $\Pi = \{A\}$ *we have*

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Diamond A, \Pi) \\ &\models \partial(A, \Pi) \vee \partial(\bigcirc \Diamond A, \Pi) \\ &\models \mathit{true} \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”}) \end{aligned}$$

Since the propositional formula is trivially true, as a minimal interpretation we have $q' = \emptyset$*. Considering that the empty conjunction is considered as true (as explained earlier), at the end of the iteration we have:*

$$\begin{aligned} q_0 &= \{\Diamond A\} \\ Q &= \{q_0, \emptyset\} \\ F &= \{\emptyset\} \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset), (q_0, \{A\}, \emptyset)\} \end{aligned}$$

- *with* $\Pi = \{\}$ *we have*

$$\begin{aligned} q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\ &\models \partial(\Diamond A, \Pi) \\ &\models \partial(A, \Pi) \vee \partial(\bigcirc \Diamond A, \Pi) \\ &\models \mathit{false} \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond \mathit{true}\text{”}) \end{aligned}$$

As a minimal interpretation we have $q' = \{\text{“}\Diamond A\text{”}, \text{“}\Diamond \mathit{true}\text{”}\}$*. Since* $\partial(\Diamond A, \epsilon) \wedge \partial(\Diamond \mathit{true}, \epsilon) = \mathit{false} \wedge \mathit{false} \neq \mathit{true}$*, we do not add* q' *to the accepting states* F*. Thus we have:*

$$\begin{aligned} q_0 &= \{\Diamond A\} \\ Q &= \{q_0, \emptyset, \{\Diamond A, \Diamond \mathit{true}\}\} \\ F &= \{\emptyset\} \\ \delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset), \\ &\quad (q_0, \{A\}, \emptyset), \\ &\quad (q_0, \{\}, \{\Diamond A, \Diamond \mathit{true}\})\} \end{aligned}$$

2. Iteration: we already analyzed $q = \{\Diamond A\}$*, so we analyze* $q = \{\Diamond A, \Diamond \mathit{true}\}$

- *with* $\Pi = \{\}$ *we have that:*

$$
\begin{aligned}
q' &\models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi) \\
&\models \partial(\Diamond A, \Pi) \wedge \partial(\Diamond true, \Pi) \\
&\models [\partial(A, \Pi) \vee \partial(\bigcirc\Diamond A, \Pi)] \wedge [\partial(true, \Pi) \vee \partial(\bigcirc\Diamond true, \Pi)] \\
&\models [\partial(A, \Pi) \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond true\text{”})] \wedge [true \vee (\text{“}\Diamond true\text{”} \wedge \text{“}\Diamond true\text{”})] \\
&\models \partial(A, \Pi) \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond true\text{”}) \\
&\models false \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond true\text{”})
\end{aligned}
$$

  As in the previous iteration, the minimal model is $q' = \{\text{“}\Diamond A\text{”}, \text{“}\Diamond true\text{”}\}$. *Hence we add a new transition* $(\{\Diamond A, \Diamond true\}, \{\}, \{\Diamond A, \Diamond true\})$.
- *with* $\Pi = \{A\}$ *the delta-expansion is the same, except for the last step, where:*

$$q' \models true \vee (\text{“}\Diamond A\text{”} \wedge \text{“}\Diamond true\text{”})$$

  The formula is always true, hence the minimal model is $q' = \emptyset$ *and we add a new transition* $(\{\Diamond A, \Diamond true\}, \{\}.\emptyset)$.

The NFA $\mathcal{A}_\varphi$ *is then composed by:*

$$
\begin{aligned}
q_0 &= \{\Diamond A\} \\
Q &= \{q_0, \emptyset, \{\Diamond A, \Diamond true\}\} \\
F &= \{\emptyset\} \\
\delta &= \{(\emptyset, \{\}, \emptyset), (\emptyset, \{A\}, \emptyset), \\
&\quad (q_0, \{A\}, \emptyset), \\
&\quad (q_0, \{\}, \{\Diamond A, \Diamond true\}) \\
&\quad (\{\Diamond A, \Diamond true\}, \{\}, \{\Diamond A, \Diamond true\}) \\
&\quad (\{\Diamond A, \Diamond true\}, \{\}, \emptyset)\}
\end{aligned}
$$

The NFA $\mathcal{A}_\varphi = \langle 2^{\{A\}}, Q, q_0, \delta, F\rangle$ *is depicted in Figure 4.3, whereas the associated* DFA *is in Figure 4.4.*

Example 4.6. *We list other examples of* $\mathcal{A}_\varphi$ *given a* LTL$_f$/LDL$_f$ *formula* φ*, obtained by Algorithm 1:*

- Conditional Response*: the* LTL$_f$ *formula* $\varphi = \Box(A \Rightarrow \Diamond B)$ *or equivalently the* LDL$_f$ *formula* $\varphi = [true^*](\langle A\rangle tt \Rightarrow \langle true^*\rangle\langle B\rangle tt)$ *translates into the automaton depicted in Figure 4.5.*
- Alternating sequence*: the* LDL$_f$ *formula* $\varphi = \langle(A; B)^*\rangle End$ *translates into the automaton depicted in Figure 4.6.*

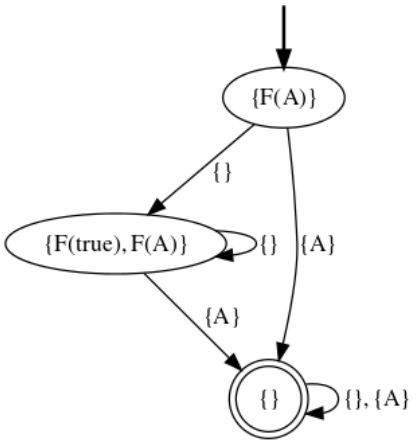


Figure 4.3. The NFA associated to $\Diamond A$. $F(A)$ stands for $\Diamond A$

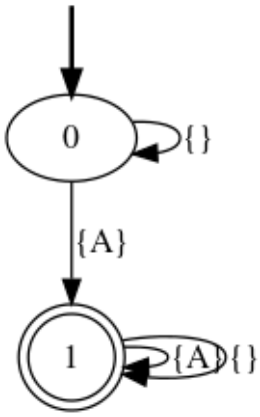


Figure 4.4. The DFA associated to $\Diamond A$

4.3 On-the-fly DFA

In this section, we describe a way to evaluate a $\text{LTL}_f/\text{LDL}_f$ formula without the need to build the entire automaton $\mathcal{A}_\varphi$. After that, we devise a new algorithm, a variant of LDL_f2NFA, that avoids the computation of the NFA, but directly translates the formula into a DFA. We provide some examples to clarify the presented topics.

4.3.1 On-the-fly $\text{LTL}_f/\text{LDL}_f$ evaluation

In this section, we describe an alternative method to evaluate a trace on a DFA without the need for constructing $\mathcal{A}_\varphi$, that we call *on-the-fly* (Brafman, De Giacomo, and Patrizi, 2018). The idea is we progress all possible states that the NFA can be in after consuming the next trace symbol and accept the trace iff, once it has been completely read, the set of possible states contains a final state.

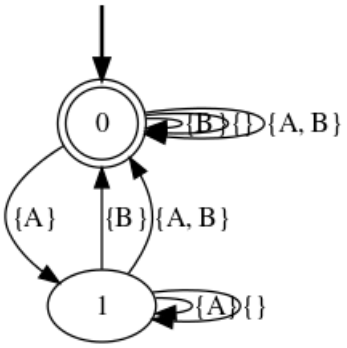


Figure 4.5. The DFA associated to $\varphi = \Box(A \Rightarrow \Diamond B)$

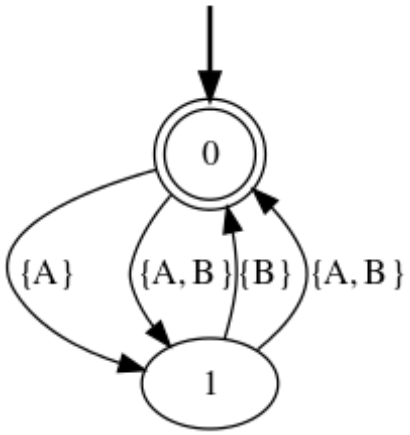


Figure 4.6. The DFA associated to $\varphi = \langle (A; B)^* \rangle End$

More formally, call a set of possible states for the NFA a macrostate, let $Q = \{q_1, \ldots, q_n\}$ be the current macrostate (initially $Q = Q_0 = \{q_0\} = \{\{\varphi\}\}$), and let Π be the next trace symbol. Then, the successor macrostate is the set Q' defined as follows:

$$Q' = \{q' | \exists q \in Q \ s.t. \ q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)\} \tag{4.5}$$

Notice that the condition $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$ is the same of the one in line 11 of Algorithm 1. Given an input trace π, we decide whether $\pi \models \varphi$ by iterating the above procedure, starting from the initial state $Q = Q_0$, and accepting π iff the last state obtained includes $\{true\}$, considering their evaluation in the empty trace (i.e. with $\partial(\psi, \epsilon)$). We denote the evaluation over the empty trace of a macrostate $Q = \{q_1, \ldots, q_n\}$ as Q^ϵ. Formally:

$$Q^\epsilon = \{\varphi | \varphi = \bigwedge_{\psi \in q_i} \partial(\psi, \epsilon)\} \tag{4.6}$$

Example 4.7. *Consider Example 4.4 with $\varphi = \Box A$, we show how the on-the-fly evaluation of traces works. At the beginning, we have that*

$$Q = Q_0 = \{\{\Box A\}\}$$

In this example, we ask the following questions:

1. *$\langle\rangle \models \varphi$? Does the empty trace $\pi = \langle\rangle$ satisfy the formula φ? We expect that the answer is yes, due to the semantics of $\Box A$. With the on-the-fly approach, we need to compute for each* NFA *state $q \in Q$ the conjunction between every $\partial(\psi, \epsilon)$, where $\psi \in q$. As said before, we consider the empty conjunction as $\{true\}$.*

   In our example, the computation gives us:

   $$Q_0^\epsilon = \{\{true\}\}$$

   because $\partial(\Box A, \epsilon) = true$. Since Q_0^ϵ contains $\{true\}$, Q_0^ϵ is an accepting state, hence $\pi \models \Box A$, as expected.

2. *$\langle\{\}\rangle \models \varphi$? This time consider the trace $\pi = \langle\{\}\rangle$ or, equivalently, $\pi = \langle\neg A\rangle$. We expect that the on-the-fly evaluation returns false, hence $\pi \not\models \varphi$. In order to answer, we need to progress the automaton on-the-fly for each element of the trace (in this case only one) and check if the last macrostate is an accepting state by using the procedure explained in the previous case. The next macrostate Q_1 by applying Equation 4.5. Actually, it is computed as we did in Iteration 1 of Example 4.4 with $\Pi = \{\}$.*

   Since no q' can be found, $Q_1 = \{\}$, which is not an accepting state since $\{true\} \notin Q_1^\epsilon$. Hence, $\pi \not\models \varphi$.

3. *$\langle\{A\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{\}\rangle$. We expect that the on-the-fly evaluation returns true, hence $\pi \models \varphi$. We proceed, as in the previous case, to compute the next macrostate Q_1 by applying Equation 4.5. Actually, it is computed as we did in Iteration 1 with $\Pi = \{A\}$. As a minimal interpretation we have both $q' = \{\Box A\}$ and $q' = \{\Box false\}$. Hence, the new macrostate is $Q_1 = \{\{\Box A\}, \{\Box false\}\}$.*

   Since there are no other symbols in the trace π to be processed, we compute $Q_1^\epsilon = \{\{true\}, \{true\}\} = \{\{true\}\}$. Since $\{true\} \in Q_1^\epsilon$, we can say that $\pi \models \varphi$.

4. *$\langle\{A\}, \{\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{A\}, \{\}\rangle$. We expect that the on-the-fly evaluation returns false, hence $\pi \not\models \varphi$. We proceed, as in the previous case, to compute the next macrostates by applying Equation 4.5. Macrostate Q_1 is the same as we have seen in Case 3. We apply again the progression rule of Equation 4.5 with $\Pi = \pi(2) = \{\}$. As a minimal interpretation we have both $q' = \{\Box A\}$ and $q' = \{\Box false\}$. Hence, the new macrostate is $Q_2 = \{\}$. as we've seen in Iteration 2 of Example 4.4.*

   Since there are no other symbols in the trace π to be processed, we compute $Q_2^\epsilon = \{\}$. Since $\{true\} \notin Q_2^\epsilon$, we can say that $\pi \not\models \varphi$.

5. *$\langle\{\}, \{A\}\rangle \models \varphi$? Consider the trace $\pi = \langle\{\}, \{A\}\rangle$. We expect that the on-the-fly evaluation returns false, hence $\pi \not\models \varphi$. The macrostate Q_1 is the same as we have seen in Case 2, i.e. $Q_1 = \{\}$. We apply again the progression rule of*

Equation 4.5 with $\Pi = \pi(2) = \{A\}$*. But this is trivially* $Q_2 = \{\}$*, by definition of the progression rule.*

Since there are no other symbols in the trace π *to be processed, we compute* $Q_2^\epsilon = \{\}$*. Since* $\{true\} \notin Q_2^\epsilon$*, we can say that* $\pi \not\models \varphi$*.*

Notice how we use the same progression of Algorithm 1, but instead of aiming to build the entire automaton, we focus only on the states that are relevant for the satisfaction of the formula, given a trace.

Example 4.8. *Analogously as we did in Example 4.7 for Example 4.4, we consider Example 4.5 with* $\varphi = \Diamond A$*, and we show how the on-the-fly evaluation of traces also works in this case. At the beginning, we have that*

$$Q = Q_0 = \{\{\Diamond A\}\}$$

In this example, we ask the following questions:

1. $\langle\rangle \models \varphi$*? Does the empty trace* $\pi = \langle\rangle$ *satisfy the formula* φ*? We expect that the answer is no, due to the semantics of* $\Diamond A$*.*

   We observe that $Q_0^\epsilon = \{\}$*, because* $\partial(\Diamond A, \epsilon) = false$*.*

   Since Q_0^ϵ *does not contain* $\{true\}$*,* Q_0^ϵ *is not an accepting state, hence* $\pi \not\models \Diamond A$*, as expected.*

2. $\langle\{\}\rangle \models \varphi$*? This time consider the trace* $\pi = \langle\{\}\rangle$ *or, equivalently,* $\pi = \langle\neg A\rangle$*. We expect that the on-the-fly evaluation returns false, hence* $\pi \not\models \varphi$*. In order to answer, we need to progress the automaton on the fly for each element of the trace (in this case only one) and check if the last macrostate is an accepting state by using the procedure explained in the previous case. The next macrostate* Q_1 *by applying Equation 4.5. Actually, it is computed as we did in Iteration 1 of Example 4.5 with* $\Pi = \{\}$*. As a minimal interpretation, we have* $q' = \{\Diamond A, \Diamond true\}$*. Hence, the new macrostate is* $Q_1 = \{\{\Diamond A, \Diamond true\}\}$*.*

   Now, $Q_1^\epsilon = \{\{false\}\}$*, which is not an accepting state since* $\{true\} \notin Q_1^\epsilon$*. Hence,* $\pi \not\models \varphi$*.*

3. $\langle\{A\}\rangle \models \varphi$*? Consider the trace* $\pi = \langle\{\}\rangle$*. We expect that the on-the-fly evaluation returns true, hence* $\pi \models \varphi$*. We proceed, as in the previous case, to compute the next macrostate* Q_1 *by applying Equation 4.5. Actually, it is computed as we did in Iteration 1 with* $\Pi = \{A\}$*. As a minimal interpretation, we have* $q' =$*. Hence, the new macrostate is* $Q_1 = \{\emptyset\}$*.*

   Since there are no other symbols in the trace π *to be processed, we compute* $Q_1^\epsilon = \{\{true\}\}$*. Since* $\{true\} \in Q_1^\epsilon$*, we can say that* $\pi \models \varphi$*.*

4. $\langle\{A\}, \{\}\rangle \models \varphi$*? Consider the trace* $\pi = \langle\{A\}, \{\}\rangle$*. We expect that the on-the-fly evaluation returns true, and so* $\pi \models \varphi$*. We proceed, as in the previous case, to compute the next macrostates by applying Equation 4.5. Macrostate* Q_1 *is the same as we have seen in Case 3. We apply again the progression rule of Equation 4.5 with* $\Pi = \pi(2) = \{\}$*, that leads to the new macrostate is* $Q_2 = \{\emptyset\}$*. Notice that* $Q_1 = Q_2$*.*

   Since there are no other symbols in the trace π *to be processed, we compute* $Q_2^\epsilon = \{\{true\}\}$*. Since* $\{true\} \in Q_2^\epsilon$*, we can say that* $\pi \models \varphi$*.*

5. $\langle\{\},\{A\}\rangle \models \varphi$? *Consider the trace* $\pi = \langle\{\},\{A\}\rangle$. *We expect that the on-the-fly evaluation returns true, hence* $\pi \models \varphi$. *The macrostate* Q_1 *is the same as we have seen in Case 2, i.e.* $Q_1 = \{\{\Diamond A, \Diamond true\}\}$. *We apply again the progression rule of Equation 4.5 with* $\Pi = \pi(2) = \{A\}$. *But this is trivially* $Q_2 = \{\emptyset\}$ *(as we've seen in Iteration 2 of Example 4.5), by definition of the progression rule.*

   Since there are no other symbols in the trace π *to be processed, we compute* $Q_2^\epsilon = \{\{true\}\}$. *Since* $\{true\} \in Q_2^\epsilon$, *we can say that* $\pi \models \varphi$.

Notice how we use the same progression of Algorithm 1, but instead of aiming to build the entire automaton, we focus only on the states that are relevant for the satisfaction of the formula, given a trace.

4.3.2 LDL$_f$2DFA: a variant of LDL$_f$2NFA

Example 4.7 and 4.8 suggest a new way to translate LTL$_f$/LDL$_f$ formulas to automata, that is a variant of LDL$_f$2NFA (Algorithm 1). We call it LDL$_f$2DFA, and directly translate a LTL$_f$/LDL$_f$ formula to a DFA, instead of first translating into a NFA and then computing the DFA by determinization.

Algorithm 2 LDL$_f$2DFA: from LTL$_f$/LDL$_f$ formula φ to DFA $\mathcal{A}_\varphi$

```
1:  input LDL_f/LTL_f formula φ
2:  output DFA A_φ = ⟨2^P, Q, Q_0, δ, F⟩          ▷ Notice: Q is a set of macrostates.
3:  Q_0 ← {{φ}}                                   ▷ the initial state of A_φ is the initial macrostate
4:  F ← ∅
5:  Q ← {Q_0}
6:  δ ← ∅
7:  if ({true} ∈ Q_0^ε) then
8:      F ← F ∪ {Q_0}
9:  while (Q or δ change) do
10:     for (Q ∈ Q, Π ∈ 2^P) do
11:         Q' ← {}
12:         for (q ∈ Q) do    ▷ Conceptually, the same loop of Algorithm 1, line 10
13:             if (q' ⊨ ⋀_(ψ∈q) ∂(ψ, Π)) then
14:                 Q' ← Q' ∪ {q'}
15:         Q ← Q ∪ {Q'}      ▷ Add new macrostate Q to the set of macrostates Q
16:         δ ← δ ∪ {(Q, Π, Q')}
17:         if ({true} ∈ Q'^ε) then
18:             F ← F ∪ {Q'}
```

The idea behind Algorithm 2 is the following: build the DFA by doing the exploration of automaton states and determinization *at the same time*. Indeed, each macrostate tracks all the possible paths (according to the trace symbols processed) of the "implicit" NFA. The computation of the next NFA state, i.e. the for-loop at line 12, works in the same way of the for-loop in line 10 of Algorithm 1. For a single macrostate Q, given a propositional interpretation Π, the operation is made for every NFA state $q \in Q$. The next macrostate Q' is then composed by all the next NFA states q'. Given the triple Q, Π, Q', we actually have a transition of the DFA $\mathcal{A}_\varphi$. Doing this operation for every macrostate and every interpretation, until convergence, will eventually lead to the exploration of every macrostate and transitions among them.

The main advantage over Algorithm 1 is that we avoid the state explosion due to the determinization procedure since we only process reachable states of the final DFA.

Example 4.9. *We consider Example 4.4 but using Algorithm 2 for translation of* $\varphi = \Box A$ *into a* DFA.

0. Before the main loop, we have:

$$\begin{aligned} Q_0 &= \{\{\Box A\}\} \\ \mathcal{Q} &= \{Q_0\} \\ \delta &= \emptyset \\ F &= \{Q_0\} \quad (\textit{because } \{\textit{true}\} \in Q_0^{\epsilon}) \end{aligned}$$

The DFA *at this stage is depicted in Figure 4.7a.*

1. Iteration: Consider the macrostate Q_0*. Consider the (unique)* NFA *state* $\{\Box A\}$*.*

With $\Pi = \{A\}$ *we generate the new macrostate* $Q' = \{\{\Box A\}, \{\Box \textit{false}\}\}$*. We add* Q' *to* $\mathcal{Q}$ *and* (Q_0, Π, Q') *to* δ*. We followed the same steps as we did in Example 4.7, Case 3.*

With $\Pi = \{\}$ *we generate the new macrostate* $Q' = \emptyset$*. We add* Q' *to* $\mathcal{Q}$ *and* (Q_0, Π, Q') *to* δ*. Since* $\{\textit{true}\} \notin \mathcal{Q}'^{\epsilon}$*, we do not add* Q' *to* F*. We followed the same steps as we did in Example 4.7, Case 2.*

The DFA *at this stage is depicted in Figure 4.7b.*

2. Iteration: We already processed $Q = \{\{\Box A\}\}$*.*

Consider the macrostate $Q = \emptyset$*. Since there exists no* $q \in Q$*, the for-loop at line 12, we add to* δ *all the transitions of the form* $(\emptyset, \Pi, \emptyset)$*, for all* $\Pi \in 2^{\mathcal{P}}$*.*

Now consider the macrostate $Q = \{\{\Box A\}, \{\Box \textit{false}\}\}$*.*

Consider $\Pi = \{A\}$*. For* $q = \{\Box A\}$ *we generate the sub-macrostate* $q' = \{\Box A\}$ *and* $q' = \{\Box \textit{false}\}$*. For* $q = \{\Box \textit{false}\}$ *we do not generate any sub-macrostate. Hence, the resulting macrostate is* $Q' = \{\{\Box A\}, \{\Box \textit{false}\}\}$*. Since* $Q' \in \mathcal{Q}$*, we only add* (Q, Π, Q') *to* δ*.*

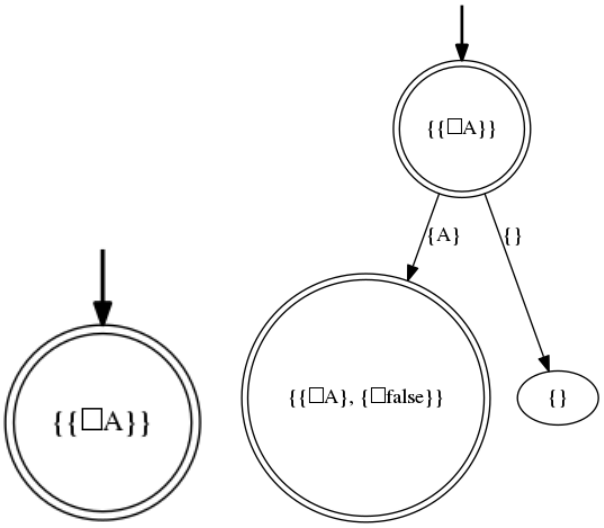
Consider $\Pi = \{\}$*. For* $q = \{\Box A\}$*, we do not generate any sub-macrostate. For* $q = \{\Box \textit{false}\}$*, we do not generate any sub-macrostate. Hence, the resulting macrostate is* $Q' = \{\}$*. Since* $Q' \in \mathcal{Q}$*, we only add* (Q, Π, Q') *to* δ*.*

Since there are no other macrostates nor propositional interpretation to process, the algorithm terminates. The final DFA *is depicted in Figure 4.7c.*

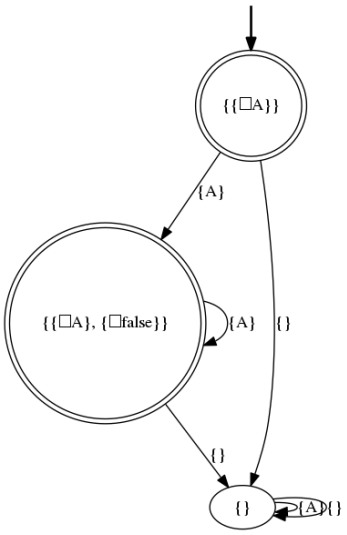
It is interesting to observe that the macrostate $Q = \{\{\Box A\}, \{\Box \textit{false}\}\}$*, where we end after reading the symbol* $\{A\}$ *from the initial state, is the set of* NFA *states (Figure 4.1) where we could ends after reading the same symbol from the initial state, namely* $\{\Box A\}$ *and* $\{\Box \textit{false}\}$*. Analogous considerations can be made for other symbols and other macrostates.*

This observation makes clearer the true meaning of Algorithm 2 with respect to Algorithm 1: that is, the macrostates keep track of all the possible evolutions of the NFA *with respect to a trace* π*. By reading all the possible symbols from every macrostate, we will eventually discover all the relevant states of the* DFA*.*

Furthermore, the minimization and trimming of the resulting automaton (shown in Figure 4.7c) yield the one shown in Figure 4.2, as an evidence of the equivalence between the two algorithms.



(a) Iteration 0 **(b)** Iteration 1



(c) Iteration 2

Figure 4.7. The automaton of Example 4.7, step by step.

Example 4.10. *We consider Example 4.5 but using Algorithm 2 for translation of* $\varphi = \Diamond A$ *into a* DFA.

0. Before the main loop, we have:

$$\begin{aligned} Q_0 &= \{\{\Diamond A\}\} \\ \mathcal{Q} &= \{Q_0, \emptyset\} \\ \delta &= \emptyset \\ F &= \emptyset \quad (\textit{because } \{\textit{true}\} \notin Q_0^\epsilon) \end{aligned}$$

The DFA *at this stage is depicted in Figure 4.8a.*

1. Iteration: Consider the macrostate Q_0*. Consider the (unique)* NFA *state* $\{\Diamond A\}$*.*

With $\Pi = \{A\}$ *we generate the new macrostate* $Q' = \{\emptyset\}$*. We add* Q' *to* $\mathcal{Q}$ *and* (Q_0, Π, Q') *to* δ*. Since* $\{true\} \in \mathcal{Q}'^\epsilon$*, we add* Q' *to* F*. We followed the same steps as we did in Example 4.8, Case 3.*

With $\Pi = \{\}$ *we generate the new macrostate* $Q' = \{\{\Diamond A, \Diamond true\}\}$*. We add* Q' *to* $\mathcal{Q}$ *and* (Q_0, Π, Q') *to* δ*. Since* $\{true\} \notin \mathcal{Q}'^\epsilon$*, we do not add* Q' *to* F*. We followed the same steps as we did in Example 4.8, Case 2.*

The DFA *at this stage is depicted in Figure 4.8b.*

2. Iteration: We already processed $Q = \{\{\Box A\}\}$*.*

Consider the macrostate $Q = \{\emptyset\}$*. Since the unique successor state of* $q = \emptyset$ *is* $q' = \emptyset$*, the next macrostate, for every symbol, is the same. Hence, we add to* δ *all the transitions of the form* $(\{\emptyset\}, \Pi, \{\emptyset\})$*, for all* $\Pi \in 2^{\mathcal{P}}$*.*

Now consider the macrostate $Q = \{\{\Diamond A, \Diamond true\}\}$*.*

Consider $\Pi = \{A\}$*. As we've seen in Case 5 of Example 4.8, the next macrostate is* $Q' = \{\emptyset\}$*, since that for* $q = \{\Diamond A, \Diamond true\}$ *the successor state is* $q' = \emptyset$*. Since* $Q' \in \mathcal{Q}$*, we only add* (Q, Π, Q') *to* δ*.*

Consider $\Pi = \{\}$*. The successor state of* $q = \{\Diamond A, \Diamond true\}$ *is* $q' = q$*. Hence, the resulting macrostate is* $Q' = \{\{\Diamond A, \Diamond true\}\}$*. Since* $Q' \in \mathcal{Q}$*, we only add* (Q, Π, Q') *to* δ*. We followed the same steps as we did in Example 4.5, Iteration 2.*

Since there are no other macrostates nor propositional interpretation to process, the algorithm terminates. The final DFA *is depicted in Figure 4.8c.*

4.4 From LTL$_f$ to FOL using Mona

In this section, we describe a different technique to transform an LTL$_f$ formula to a DFA, which passes through a first-order formalism and then uses the `Mona` tool.

4.4.1 Reduction to FOL

For an LTL$_f$ formula, we can compute an equivalent formula in monadic first-order logic on finite linearly ordered domains (FO[<], or FOL for simplicity), firstly illustrated in (De Giacomo and Vardi, 2013).

Specifically, we consider a first-order language that is formed by the two binary predicates *succ* and $<$ (which we use in the usual infix notation) plus equality, a

unary predicate for each symbol in $\mathcal{P}$ and two constants 0 and *last*. Then we restrict our interest to finite linear ordered FOL interpretation, which are FOL interpretations of the form $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, where the domain is $\Delta^I = \{0, \ldots, n\}$ with $n \in \mathbb{N}$, and the interpretation function $\cdot^{\mathcal{I}}$ interprets binary predicates and constants in a fixed way:

- $succ^{\mathcal{I}} = \{(i, i+1) \mid i \in \{0, \ldots, n-1\}\}$,
- $<^{\mathcal{I}} = \{(i, j) \mid i, j \in \{0, \ldots, n\} \text{ and } i < j\}$,
- $=^{\mathcal{I}} = \{(i, i) \mid i \in \{0, \ldots, n\}\}$,
- $0^{\mathcal{I}} = 0$ and $last^{\mathcal{I}} = n$.

In fact, *succ*, $=$, 0 and *last* can all be defined in terms of $<$. Specifically:

- $succ(x, y) \doteq (x < y) \wedge \neg\exists z.x < z < y$;
- $x = y \doteq \forall z.x < y \equiv y < z$;
- 0 can be defined as that x such that $\neg\exists y.succ(y, x)$, and *last* as that x such that $\neg\exists y.succ(x, y)$.

For convenience, we keep these symbols in the language. Also, we use the usual abbreviation $x \leq y$ for $x < y \vee x = y$.

In spite of the obvious notational differences, it is easy to see that finite linear ordered FOL interpretations and finite traces interpretations are isomorphic. Indeed, given a finite trace π we define the corresponding finite FOL interpretation $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$ as follows: $\Delta^I = \{0, \ldots, last\}$ (with $last = length(\pi) - 1$); with the obvious, predefined predicates and constants interpretation, and, for each $A \in \mathcal{P}$, its interpretation is $A^{\mathcal{I}} = \{i | A \in \pi(i)\}$. Conversely, given a finite linear ordered FOL interpretation $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, with $\Delta^I = \{0, \ldots, n\}$, we define the corresponding trace π as follows: $length(\pi) = n + 1$; and for each position $0 \leq i \leq last$, with $last = n$, we have $\pi(i) = A \mid i \in A^{\mathcal{I}}$.

We can then use a translation function $fol(\varphi, x)$ that, given an LTL$_f$ formula φ and a variable x, returns a corresponding FOL formula open in x. We define $fol()$ by induction on the structure of the LTL$_f$ formula:

- $fol(A, x) = A(x)$
- $fol(\neg\varphi, x) = \neg fol(\varphi, x)$
- $fol(\varphi \wedge \varphi', x) = fol(\varphi, x) \wedge fol(\varphi', x)$
- $fol(\bigcirc\varphi, x) = \exists y.succ(x, y) \wedge fol(\varphi, y)$
- $fol(\varphi\, \mathcal{U}\, \varphi', x) = \exists y.x \leq y \leq last \wedge fol(\varphi', y) \wedge \forall z.x \leq z < y \rightarrow fol(\varphi, z)$

Theorem 4.11 ((De Giacomo and Vardi, 2013))**.** *Given trace π and a corresponding finite linear ordered* FOL *interpretation $\mathcal{I}$, we have $\pi, i \models \varphi$ iff $\mathcal{I}, [x/i] \models fol(\varphi, x)$, where $[x/i]$ stands for a variable assignment that assigns to the free variable x of $fol(\varphi, x)$ the value i.*

4.4.2 Weak monadic Second-order theory of 1 Successor (WS1S)

It has been known since 1960 that the class of regular languages is linked to decidability questions in formal logic. In particular, WS1S (*Weak monadic Second-order theory of 1 Successor*) is decidable through the automaton-logic connection, which can be simply stated: the set of satisfying interpretations of a subformula is represented by a finite-state automaton (Büchi, 1960a; Elgot, 1961). WS1S thus acts as a notation for regular languages, just as regular expressions do.

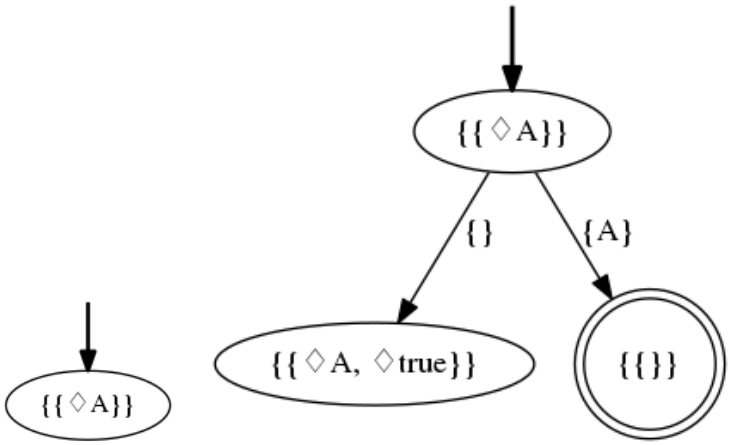
The automaton for a formula can be calculated by a simple induction scheme, where logical connectives correspond to classic automata-theoretic operations such as product and subset constructions, similarly as presented in Chapter 2 but for LDL$_f$. Validity and unsatisfiability of the formula can be determined and satisfying examples and counter-examples can be constructed by analyzing the associated automaton. Despite its name, WS1S is a simple and natural notation. Being a variation of first-order logic, WS1S is a formalism with quantifiers and boolean connectives. Its interpretation, however, is tied to arithmetic, somewhat weakened to keep the formalism decidable. In WS1S, first-order variables denote natural numbers, which can be compared and subjected to addition with constants. WS1S also allows second-order variables while remaining decidable; each such variable is interpreted as a finite set of numbers.

WS1S can be used to express problems ranging from hardware verification to formal linguistics. Unfortunately, the space and time requirements for translating formulas to automata have been shown to be non-elementary (i.e., bounded from below by a stack of exponentials whose height is proportional to the length of the formula) (Meyer, 1975). Thus, the decidability property has for many years been considered intractable for practical use.

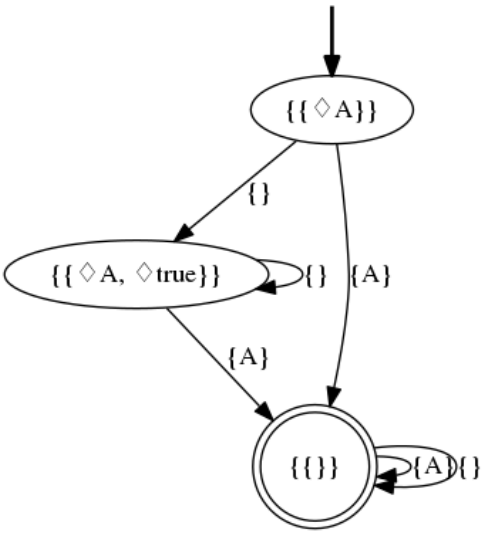
One of the major implementations available of such translation is the `Mona` tool (Henriksen, Jensen, et al., 1995), which will be described more in detail in Chapter 7. The work (Zhu, Tabajara, Li, et al., 2017) employs the reduction from LTL$_f$ to FOL and then uses `Mona` to get the minimum DFA.

4.5 Summary

In this chapter, we reported the main results on the relationships between LTL$_f$/LDL$_f$ formulas and finite automata. We started by showing how an LTL$_f$/LDL$_f$ formula can be transformed in linear time to an alternating finite automata. Then, we saw how we can provide direct transformations by determinizing the AFA on-the-fly; first from LTL$_f$/LDL$_f$ to NFA, and then from LTL$_f$/LDL$_f$ to DFA. Finally, we described another known approach that passes through the FOL formalism.



(a) Iteration 0 **(b)** Iteration 1



(c) Iteration 2

Figure 4.8. The automaton of Example 4.8, step by step.

Part II

Compositional Automata Construction

Chapter 5

Compositional approach

The translation from temporal logic to automata is the workhorse algorithm of several techniques in computer science and AI, such as reactive synthesis, reasoning about actions, FOND planning with temporal specifications, and reinforcement learning with non-Markovian rewards, just to name a few. Unfortunately, as we have seen in Chapter 4, the problem is computationally intractable, requiring the implementation of several heuristics to make it usable in practice. In this chapter, following the recent interest in temporal logic formalisms over finite traces, we present one of the main contributions of the thesis. We describe a *compositional approach* for dealing with translations of Linear Temporal Logic and Linear Dynamic Logic (LDL$_f$) on finite traces into Deterministic Finite Automata DFA. That is, we inductively transform each LTL$_f$/LDL$_f$ subformula into a DFA, and combine them through automata operators. By relying on efficient semi-symbolic automata representations, we empirically show the effectiveness of our approach and the competitiveness with similar tools. Moreover, this is the first work that provides a scalable and practical tool supporting the translation to DFA not only for LTL$_f$ but also for full LDL$_f$.

The rest of the chapter is structured as follows:

- In Section 5.1, we introduce the state-of-the-art techniques and implementations of the transformation from LTL$_f$ to DFAs.
- In Section 5.2, we present the details of our compositional approach. We provide a formalization, we prove the correctness and analyze its computational complexity.
- In Section 5.3, we present several examples of the translation approach.
- In Section 5.4 summarizes the content of the chapter and we discuss the contributions and potential extension of the work.

The contents of this chapter have been published in the conference paper (De Giacomo and Favorito, 2021).

5.1 Introduction

As we have seen in Chapter 3, reasoning over LTL$_f$/LDL$_f$ is usually done by relying on automata theory. In particular, from a LTL$_f$/LDL$_f$ formula φ, we can build a deterministic finite automaton (DFA) $\mathcal{A}_\varphi$, whose alphabet is the set of propositional interpretations $\mathcal{P}$ of φ, that is semantically equivalent to the original formula (De

Giacomo and Vardi, 2013; De Giacomo and Vardi, 2015). The computational complexity of such translation has been shown to be doubly exponential time in the worst case, and indeed $\mathcal{A}_\varphi$ can be double-exponentially larger than the original formula φ. Nevertheless, in most cases the resulting DFA is actually manageable, a phenomenon often observed when determinization is applied to automata finite words (Tabakov and Vardi, 2005). This puts working in the finite traces in sharp contrast with working with infinite ones, which are hampered by the notorious intractability of determinization of nondeterministic Büchi automata (Fogarty et al., 2015).

One of the ingredients of the translation from such logic to DFAs is the `Mona` tool (Henriksen, Jensen, et al., 1995; Klarlund, 1997; Klarlund and Møller, 2001). The tool implements the translation from First-Order Logic (FO) and Monadic Second-Order Logic on finite strings (MSO) to deterministic finite automata. Thanks to its novel and efficient semi-symbolic representation, still explicit in the state space's representation but symbolic in the transitions', `Mona` has become widely used in the research community. One of the best practical implementation of the translation from LTL_f to DFA, proposed by (Zhu, Tabajara, Li, et al., 2017). Their tool `Syft` encodes LTL_f formulae into First-Order Logic formulae, represented as `Mona` programs, and uses `Mona` to perform the actual translation. The `Mona` output is then post-processed to produce a fully symbolic representation (i.e. both in the state space and in the transitions) to perform LTL_f synthesis. A more recent work (Bansal et al., 2020) proposed a hybrid approach to the problem of DFA construction from LTL_f formulae: first, they decompose the outermost conjunction in φ, where φ is assumed to be in the form $\varphi = \bigwedge_{i=1}^{n} \varphi_i$, in n-subformulae $\varphi_1 \ldots, \varphi_n$. Then, they transform each φ_i into DFAs $\mathcal{A}_{\varphi_i}$ in explicit-state representation using `Mona`. Finally, they start doing the product between all the automata $\mathcal{A}_{\varphi_i}$; if at some point the size of the partial automaton becomes too large and exceeds a user-defined threshold, the approach converts all the explicit-state automata in symbolic representation and continues with the products, though forgoing minimization. In this way the tool is able to scale even in the case the automaton becomes prohibitively large to be represented explicitly, although not producing a minimal automaton anymore in this case. Both tools in (Zhu, Tabajara, Li, et al., 2017) and in (Bansal et al., 2020) perform much better than state of the art tools, such as SPOT (Duret-Lutz et al., 2016), which implement procedures to translate LTL formulae to automata on infinite words, and can also be used for LTL_f by exploiting its encoding into LTL (De Giacomo and Vardi, 2013). They implemented a tool called `Lisa` and `LisaSynt`, for DFA translation and synthesis, respectively.

Observe that both tools make use of the translation of FO into DFA, provided by `Mona`, which is nonelementary[1] in the worst case, due to the necessity of multiple determinizations (each exponential in the worst case) and projections (which introduces nondeterminism) needed to handle quantifiers and negations. Still, this non-elementariness does not show in practice (again for the phenomenon of determinization of automata on finite words mentioned above).

Here we take a step further from the compositional approach proposed in (Bansal et al., 2020). In particular, the contribution of this chapter is a fully compositional approach to handle both LTL_f formulae and LDL_f formulae. That is, we do not make any assumption on the structure of the formula, as done by Bansal et al. which stops the decomposition step at the outermost conjunction. We process all the

[1] In computational complexity theory, a *nonelementary problem* is a problem that is not a member of the ELEMENTARY class, i.e., the computational time cost of such problems has an unbounded number of exponentiations.

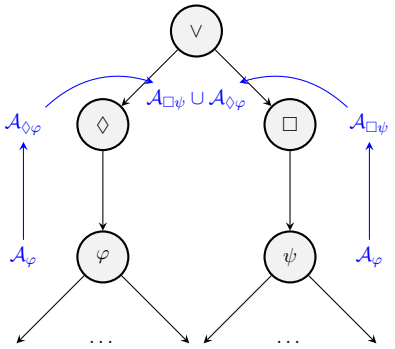


Figure 5.1. The compositional technique for $\Diamond\varphi \vee \Box\psi$, at a glance. It starts translating the subformulas that are deepest in the syntax tree, and then compose them backward according to the logic operators.

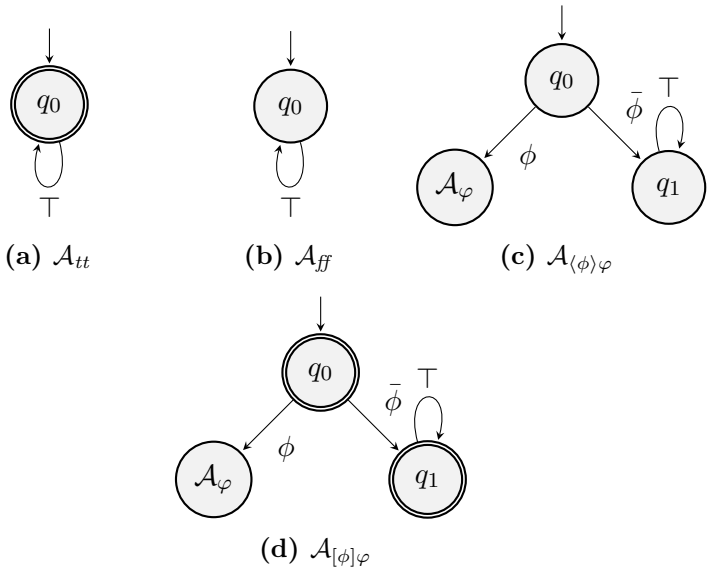
subformulae recursively up to the leaves of the syntax tree, and then we compose the partial DFAs of the subformulae using common operations over automata (e.g. union, intersection, concatenation), according to the $\text{LTL}_f/\text{LDL}_f$ operator being processed.

The contribution of this technique is both theoretical and practical. On the theoretical side, we observe that so far the theory of the correspondence between $\text{LTL}_f/\text{LDL}_f$ and automata theory relied on the transformation of $\text{LTL}_f/\text{LDL}_f$ formulae into alternating automata on finite words (AFA), which can be eventually transformed into nondeterministic finite automata (NFA), and in turn determinized into DFAs (De Giacomo and Vardi, 2013). Instead, we provide a sound and complete technique to directly transform a formula into a DFA. Despite the worst-case complexity of such technique is again nonelementary, as `Mona`'s, we show that it has several practical advantages with respect to the previous ones, primarily due to the possibility of applying aggressive minimization to the partial automata, which has already been argued to be indispensable for scalability (Klarlund and Møller, 2001; Zhu, Tabajara, Pu, et al., 2021). On the practical side, in Chapter 6 and Chapter 7, we describe an implementation that employs such a compositional technique and showing its competitiveness with existing tools (Bansal et al., 2020; Henriksen, Jensen, et al., 1995). The tool can be used both for $\text{LTL}_f/\text{LDL}_f$-to-DFA construction and as a $\text{LTL}_f/\text{LDL}_f$ synthesis tool. Crucially, this is the first work that provides a scalable and practical tool supporting the translation to DFA and synthesis not only for LTL_f but also for full LDL_f. The tool will be described more in detail in Chapter 7.

This chapter is an extended version of the conference paper in (De Giacomo and Favorito, 2021).

5.2 Compositional Translation

In this section, we describe the technique inductively translate each basic $\text{LTL}_f/\text{LDL}_f$ formula and operators over them into (minimal) DFAs. We call the technique "compositional" due to its focusing on smaller subproblems and in the successive composition of partial results. We provide direct transformations from LDL_f to automata; for what concerns LTL_f, we apply the transformation rules explained in the "Preliminaries" section. Finally, we will provide a theoretical analysis of the technique. In Figure 5.1, it is depicted at a high level how the compositional translation works.



(a) $\mathcal{A}_{tt}$ **(b)** $\mathcal{A}_{ff}$ **(c)** $\mathcal{A}_{\langle\phi\rangle\varphi}$

(d) $\mathcal{A}_{[\phi]\varphi}$

Figure 5.2. DFAs of elementary LDL$_f$ formulae.

5.2.1 The Technique

In what follows, we describe the transformation for each elementary formula and operator of LDL$_f$ into an equivalent DFA. The approach is "bottom-up": it computes the DFA of the deepest subformulae, and combines the partial results depending on the LDL$_f$ operator under transformation. This is in contrast with the previous techniques known in the literature that are "top-down": they proceed from the root operator of the formula in order to compute the next states (see e.g. LDL$_f$2NFA in (De Giacomo and Vardi, 2013; De Giacomo and Vardi, 2015; Brafman, De Giacomo, and Patrizi, 2018)).

tt and ff: the logical true formula *tt* is equivalent to a DFA with an unique accepting state and a loop that accepts all symbols (Figure 5.2a). In other words, it is the minimal automaton that accepts the language Σ^*. Its dual, *ff*, is the automaton of the empty language (Figure 5.2b).

$\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\varphi$: The boolean operations over LDL$_f$ formulae are processed with the corresponding boolean operations over automata. For conjunction and disjunction, we use the product construction with respectively conjunction or disjunction of states as accepting conditions; for negation, we use the complementation of automata. The output of these operations might require a further minimization and completion step.

$\langle\phi\rangle\varphi$: the diamond formula with a propositional formula as regular expression is equivalent to the automaton in Figure 5.2c. With the empty trace, the run fails. Otherwise, the next input symbol of the trace is read; if it satisfies ϕ, then the run proceeds with the simulation of the automaton associated to φ (starting from the state labelled with $\mathcal{A}_\varphi$), else the run fails and goes to the sink state. Observe that the operation might require a further minization step, even if $\mathcal{A}_\varphi$ is minimal; e.g. take $\varphi = ff$ as example.

$[\phi]\varphi$: the box formula with a propositional formula as regular expression is equivalent to the automaton in Figure 5.2d. With the empty trace, the run succeeds. Otherwise, the next input symbol of the trace is read; if it satisfies ϕ, then the run

proceeds with the simulation of the automaton associated to φ (starting from the state labelled with $\mathcal{A}_\varphi$), else the run succeeds and goes to the sink accepting state. Observe that the operation might require a further minimization step, even if $\mathcal{A}_\varphi$ is minimal; e.g. take $\varphi = tt$ as example.

$\langle\psi?\rangle\varphi$ and $[\psi?]\varphi$: The formulae can be reduced to $\psi \wedge \varphi$ and $\neg\psi \vee \varphi$, respectively.

$\langle\rho_1;\rho_2\rangle\varphi$ and $[\rho_1;\rho_2]\varphi$: Both formulae are reducible to $\langle\rho_1\rangle\langle\rho_2\rangle\varphi$ and $[\rho_1][\rho_2]\varphi$, respectively.

$\langle\rho_1+\rho_2\rangle\varphi$ and $[\rho_1+\rho_2]\varphi$: These formulae can be reduced to $\langle\rho_1\rangle\varphi \vee \langle\rho_2\rangle\varphi$ and $[\rho_1]\varphi \wedge [\rho_2]\varphi$, respectively.

$\langle\rho^*\rangle\varphi$ and $[\rho^*]\varphi$: It is enough to translate $\langle\rho^*\rangle\varphi$ and get the other by the duality of the diamond operator, i.e. $[\rho^*]\varphi \equiv \neg\langle\rho^*\rangle\neg\varphi$. Hence, we will only consider $\langle\rho^*\rangle\varphi$. To compute the automaton $\mathcal{A}_{\langle\rho^*\rangle\varphi}$, we first consider the case in which ρ does not contain any test. In this case, we have that the automaton $\mathcal{A}_\rho$ of ρ, is equivalent to the automaton of $\langle\rho\rangle end$, i.e. $\mathcal{A}_\rho = \mathcal{A}_{\langle\rho\rangle end}$, as the semantics of LDL$_f$ formulae of the form $\langle\rho\rangle end$ is the same of RE$_f$ formulae ρ. Hence, the automaton $\mathcal{A}_{\langle\rho\rangle end}$ can be computed using the well-known construction of DFA from regular expressions (See, e.g. (Hopcroft, Motwani, and Ullman, 2006)). Then, we compute the Kleene closure of $\mathcal{A}_\rho$, $\mathcal{A}_{\rho^*}$. Finally, we concatenate $\mathcal{A}_{\rho^*}$ and $\mathcal{A}_\varphi$ to obtain the desired automaton. This approach can be generalized to handle tests as well in some cases, but not always since it could happen that the verification of a test $\psi?$ could take more steps than the regular expression ρ itself. When this happens it is no longer true that $\mathcal{A}_\rho$ and $\mathcal{A}_{\langle\rho\rangle end}$ are equivalent since the presence of end in the second one would stop the evaluation of the test $\psi?$ too early, changing the semantics of the formula. Hence when we cannot guarantee that this does not happen, we simply fall back to using the classical algorithm that computes the AFA from $\langle\rho^*\rangle\varphi$ (De Giacomo and Vardi, 2013; Brafman, De Giacomo, and Patrizi, 2018), with the only difference that we recursively pre-compute the DFA $\mathcal{A}_\psi$ for each test $\psi?$ and the DFA $\mathcal{A}_\varphi$ for φ, and whenever we go to state $\psi?$ or φ in the AFA of $\langle\rho^*\rangle\varphi$ we actually go to the initial state of the DFAs $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ Then we transform the AFA into a NFA as usual and then determinize it to obtain the desired DFA. The reason why we adopted two different approaches for $\langle\rho^*\rangle\varphi$ is that the case when ρ does not contain tests allows us to better decompose the problem. Intuitively, this happens because of the lack of universal transitions due to the absence of the test expressions in ρ.

To summarize, in order to compute the DFA $\mathcal{A}_\varphi$ equivalent to an LDL$_f$ formula φ, recursively apply the transformations stated above, one for each syntactic construct of the formula.

5.2.2 Analysis

Now we analyze the technique, proving correctness, termination, and running time complexity.

Theorem 5.1. *(Correctness) Let φ be an* LDL$_f$ *formula and $\mathcal{A}_\varphi$ the corresponding* DFA*. Then for every* LTL$_f$ *-interpretation π we have that $\pi \models \varphi \iff \pi \in \mathcal{L}(\mathcal{A}_\varphi)$.*

Proof. We prove a more general statement, that is $\forall i.\pi, i \models \varphi \iff \pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_\varphi)$. Clearly, the claim of the theorem corresponds to the case $i = 0$. For $i > 0$, we proceed by induction on the structure of φ.

- $\varphi = tt$. Then, on the one hand, $\pi, i \models tt$. On the other hand, $\pi(i, length(\pi)) \in \mathcal{L}(\varphi_{tt})$, where $\mathcal{L}(\varphi_{tt}) = \Sigma^*$.

- $\varphi = \mathit{ff}$. Then, on the one hand, $\pi, i \not\models \mathit{ff}$. On the other hand, $\pi(i, \mathit{length}(\pi)) \notin \mathcal{L}(\varphi_{\mathit{ff}})$, where $\mathcal{L}(\varphi_{\mathit{ff}}) = \emptyset$.
- $\varphi = \neg\varphi'$. Then, $\pi, i \models \varphi'$, and, by definition, $\pi, i \not\models \varphi$. By structural induction, we have that $\pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ and so $\pi(i, \mathit{length}(\pi)) \notin \mathcal{L}(\mathcal{A}_{\neg\varphi})$, hence $\pi(i, \mathit{length}(\pi))$ is not accepted by $\mathcal{A}_\varphi = \overline{\mathcal{A}_{\varphi'}}$.
- $\varphi = \varphi_1 \wedge \varphi_2$. We have both $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$. By structural induction, we then have that $\pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_1})$ and $\pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_2})$, which is the condition of acceptance for $\pi(i, \mathit{length}(\pi))$ on $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cap \mathcal{A}_{\varphi_2}$.
- $\varphi = \varphi_1 \vee \varphi_2$. We have either $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$. By structural induction, we then have that $\pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_1})$ or $\pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_2})$, which is the condition of acceptance for $\pi(i, \mathit{length}(\pi))$ on $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cup \mathcal{A}_{\varphi_2}$.
- $\varphi = \langle\rho\rangle\varphi'$. We proceed by induction on ρ, and we show that for every φ', $\pi, i \models \langle\rho\rangle\varphi' \iff \pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho\rangle\varphi'})$.
  - $\rho = \phi$. We have that $\pi, i \models \langle\phi\rangle\varphi'$ iff $(i, i+1) \in \mathcal{R}(\phi, \pi)$ and $\pi, i+1 \models \varphi$. Notice also that $\mathcal{A}_{\langle\phi\rangle\varphi'}$ is of the form shown in Figure 5.2c. Observe that if $i \geq \mathit{length}(\pi)$ then $\pi, i \models \langle\phi\rangle\varphi'$ is false, and indeed the empty trace $\pi(i, \mathit{length}(\pi)) = \epsilon$ is not accepted by $\mathcal{A}_{\langle\phi\rangle\varphi'}$. If $i < \mathit{length}(\pi)$, then $\pi, i \models \langle\phi\rangle\varphi'$ iff $\pi(i) \models \phi$ and $\pi, i+1 \models \varphi'$, which is iff the transition from q_0 and $\mathcal{A}_\varphi$ is taken, and then $\pi(i+1, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$.
  - $\rho = \psi?$. Observe that $\langle\psi?\rangle\varphi' \equiv \psi \wedge \varphi'$, thus this case is addressed by applying the same reasoning as the one for conjunction.
  - $\rho = \rho_1 + \rho_2$. Observe that $\langle\rho_1 + \rho_2\rangle\varphi' \equiv \langle\rho_1\rangle\varphi' \vee \langle\rho_2\rangle\varphi'$, thus this case is addressed by applying the same reasoning as the one for disjunction.
  - $\rho = \rho_1; \rho_2$. Observe that $\langle\rho_1; \rho_2\rangle\varphi' \equiv \langle\rho_1\rangle\langle\rho_2\rangle\varphi'$. By induction on ρ_2 we have that $\pi, i \models \langle\rho_2\rangle\varphi' \iff \pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho_2\rangle\varphi'})$. By induction on ρ_1, we have that for all ψ, $\pi, i \models \langle\rho_1\rangle\psi \iff \pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho_1\rangle\psi})$. By replacing ψ with $\langle\rho_2\rangle\varphi'$, and considering that the automaton $\mathcal{A}_{\langle\rho_1;\rho_2\rangle\varphi'}$ is by definition $\mathcal{A}_{\langle\rho_1\rangle\langle\rho_2\rangle\varphi'}$, the thesis follows.
  - $\varphi = \langle\rho^*\rangle\varphi'$. We first consider the case where ρ does not contain tests. We prove this case by induction on $n = \mathit{length}(\pi(i, \mathit{length}(\pi)))$.

    First, assume $n = 0$. This implies that $i \geq \mathit{length}(\pi)$, and hence $\pi(i, \mathit{length}(\pi)) = \epsilon$, i.e. is the empty trace. Since we are out-of-bounds and no propositional formulae can be executed, and the only case that matters is the one with zero repetition of ρ in ρ^*: $\pi, i \models \langle\rho^*\rangle\varphi'$ holds iff $\pi, i \models \varphi'$. By structural induction, $\pi, i \models \varphi'$ holds iff $\mathcal{A}_{\varphi'}$ accepts $\pi(i, \mathit{length}(\pi)) = \epsilon$. Now, consider the construction of $\mathcal{A}_{\langle\rho^*\rangle\varphi'}$. It is the concatenation of $\mathcal{A}_{\langle\rho^*\rangle end}$ and $\mathcal{A}_{\varphi'}$. Since $\mathcal{A}_{\rho^*}$ accepts the empty trace by construction (it is the Kleene closure of $\mathcal{A}_{\langle\rho\rangle end}$), $\mathcal{A}_{\langle\rho^*\rangle\varphi'}$ accepts the empty trace iff $\mathcal{A}_{\varphi'}$ accepts the empty trace.

    Now, assume that $n > 0$ and the claim holds for every $n' < n$. From the semantics of $\langle\rho^*\rangle\varphi$, we have that $\pi, i \models \langle\rho^*\rangle\varphi'$ iff exists $j \geq i$ s.t. either $j = i$ and $\pi, j \models \varphi'$ or there exists $j > i$ such that $(i, k) \in \mathcal{R}(\rho, \pi)$ and $(k, j) \in \mathcal{R}(\rho^*, \pi)$ and $\pi, j \models \varphi'$, with $k > i$. We want to prove that for every φ', $\pi, i \models \langle\rho^*\rangle\varphi'$ iff $\pi(i, \mathit{length}(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle\varphi'})$. We distinguish two cases; one in which there are zero repetitions of ρ ($j = i$), and the other when there are one or more ($j > i$).

In case there are zero repetitions, we have that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle\varphi'})$ iff $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ by construction, and that $\pi, i \models \langle\rho^*\rangle\varphi'$. iff $\pi, i \models \varphi'$ by the semantics, so now we need to prove that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ iff $\pi, i \models \varphi'$, but this is true by structural induction.
In the other case, $j > i$, we have one or more repetitions of ρ. By construction, there exists a $k > i$ such that $\pi(i, k) \in \mathcal{L}(\mathcal{A}_{\langle\rho\rangle end})$, $\pi(k, j) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle end})$, and $\pi(j, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$. We have that $\pi(i, k) \in \mathcal{L}(\mathcal{A}_{\langle\rho\rangle end})$ iff $(i, k) \in \mathcal{R}(\rho, \pi)$ by construction, $\pi(k, j) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle end})$ iff $(k, j) \in \mathcal{R}(\rho^*, \pi)$ by induction on the length of the trace, and $\pi(j, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ iff $\pi, j \models \varphi'$ by structural induction. Combining the above equivalences, we get the thesis.
Let us now consider the case in which ρ instead contains tests. Let $\psi_1?, \ldots, \psi_n?$ be all tests in ρ. Let $\mathcal{A}_{\psi_i}$ be the DFA associated to the test $\psi_i?$. Note that both these DFAs as well as $\mathcal{A}_{\varphi'}$ are correct by structural induction. Then we compute the AFA $\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'}$ as in (De Giacomo and Vardi, 2013; Brafman, De Giacomo, and Patrizi, 2018), but with the difference that states of the form $\psi_i?$ or φ' are replaced by the initial states of $\mathcal{A}_{\psi_i}$ and $\mathcal{A}_{\varphi'}$, respectively. Moreover, the other states and transitions of these DFAs are added to the states and transitions of $\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'}$. Then, we have that $\pi, i \models \langle\rho^*\rangle\varphi$ $\pi(i, length(\phi)) \in \mathcal{L}(\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'})$, and since from $\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'}$ we can obtain an equivalent DFA we get the thesis. □

It is also of interest to make some observations on the intermediate automata generated by the technique. The computation of DFAs of simple formulae tt, ff, $\langle\phi\rangle\varphi$ and $[\phi]\varphi$, given the DFA for φ, can be done in constant time, since they don't depend directly on the size of ϕ nor φ. Negation consists in changing accepting states to rejecting states and vice versa. The other boolean operations are translated using products of DFAs, which are polynomial. The computation of $\mathcal{A}_{\langle\rho\rangle\varphi}$ without the occurrence of the * operator can be handled reducing recursively to the previous cases without introducing any non-determinism. The occurrence of the * instead prevents us to reduce to the previous cases, and introduces non-determinism due to the Kleene closure and the concatenation operations, and hence exponential steps to determinize the resulting automaton (Maslov, 1970; Yu, Zhuang, and Salomaa, 1994). More precisely, let us consider a sub-formula $\langle\rho^*\rangle\varphi$. If ρ does not contain tests[2] and does not contains star operators, then computing the DFA $\mathcal{A}_{\langle\rho\rangle end}$ is polynomial, and computing the DFA for the Kleene closure, $\mathcal{A}_{\langle\rho^*\rangle end}$, is exponential w.r.t the size of $\mathcal{A}_{\langle\rho\rangle end}$. As it is exponential doing the concatenation with $\mathcal{A}_{\varphi}$, but w.r.t. the size of $\mathcal{A}_{\varphi}$ hence the total contribution is one exponential. If ρ contains star operators, then for the arguments above those sub-expressions already contribute with an arbitrary number of exponentials, and the outermost star contributes with another exponential for the same arguments. If ρ contains tests, then we switch to the AFA construction which contributes with a double-exponential cost due to transformation to NFA and to determinization to obtain the DFA.

Summarizing, any nested star operation gives, in the worst case, an exponential blow-up and hence is nonelementary. Although this may sound discouraging, we observe that practical tools like `Mona` (Henriksen, Jensen, et al., 1995) implement a nonelementary technique; yet, they perform very well in practice. We show that also

[2] Or we are guaranteed that the test is completed within the part of the word scanned by ρ.

our implementation of the technique is competitive with `Mona` and other tools. Also, observe that in our implementation, like in `Mona`, we *aggressively minimize* the partial DFA obtained after each compositional step. Since the cost of DFA minimization for automata with explicit-state representations can be done in $\mathcal{O}(n \log n)$ (Hopcroft, 1971), this does not worsen the complexity of the technique, while in practice enhances it substantially because often the minimal DFA obtained from an NFA is of size comparable to the NFA itself, instead of being exponential in it. In practice, this subset construction with only reachable subset states is often linear, not exponential as it may be feared, as observed in the literature, see e.g. (Basin and Klarlund, 1998).

In any case, since the technique is correct (c.f., Theorem 5.1), by the uniqueness of minimal DFAs, the returned DFA (once minimized) is at most double-exponentially larger than the LDL_f formula (De Giacomo and Vardi, 2013; De Giacomo and Vardi, 2015; Brafman, De Giacomo, and Patrizi, 2018).

5.3 Examples

In this section, we show several examples describing the compositional technique.

Example 5.2. *Let $\varphi = \langle a + b \rangle \langle c; d \rangle tt$ be an LDL_f formula. Note that, according to the translation rules, it can be first translated into:*

$$\varphi' = \langle a \rangle \langle c \rangle \langle d \rangle tt \vee \langle b \rangle \langle c \rangle \langle d \rangle tt$$

with $\varphi \equiv \varphi'$.

We start from processing the right-hand side $\langle a \rangle \langle c \rangle \langle d \rangle tt$. We start from the subformula tt, getting $\mathcal{A}_{tt}$ (Figure 5.3):

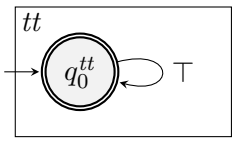


Figure 5.3. The automaton for tt.

Then, we process $\langle d \rangle tt$ by using the automaton template shown in Figure 5.2c, yielding $\mathcal{A}_{\langle d \rangle tt}$ (Figure 5.4):

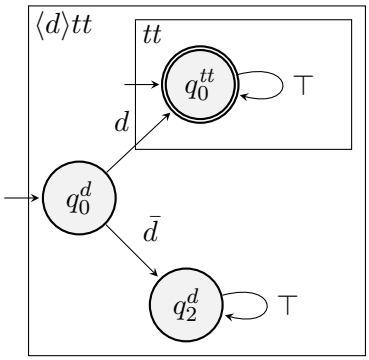


Figure 5.4. The automaton for $\langle d \rangle tt$.

We proceed similarly for $\langle c\rangle\langle d\rangle tt$ and $\langle b\rangle\langle c\rangle\langle d\rangle tt$, yielding automata $\mathcal{A}_{\langle c\rangle\langle d\rangle tt}$ and $\mathcal{A}_{\langle b\rangle\langle c\rangle\langle d\rangle tt}$, depicted in Figure 5.5 and 5.6, respectively:

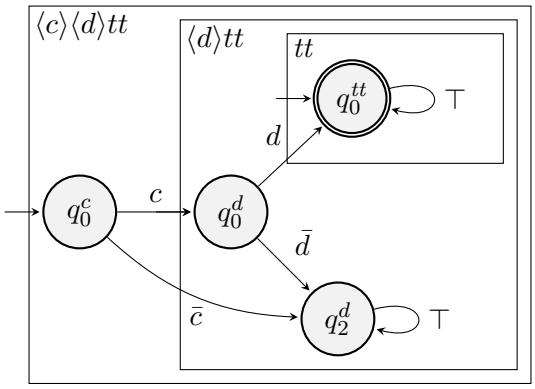


Figure 5.5. The automaton for $\langle c\rangle\langle d\rangle tt$.

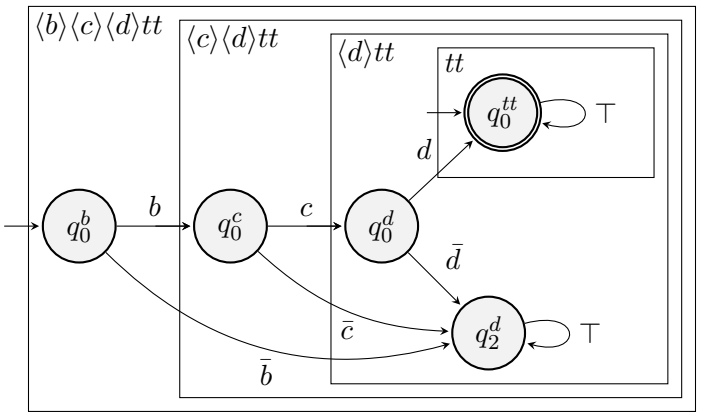


Figure 5.6. The automaton for $\langle b\rangle\langle c\rangle\langle d\rangle tt$.

For the left-hand side of the formula, note that the resulting automaton is the same of $\mathcal{A}_{\langle b\rangle\langle c\rangle\langle d\rangle tt}$ except for the last step, where b has to be replaced with a (Figure 5.7):

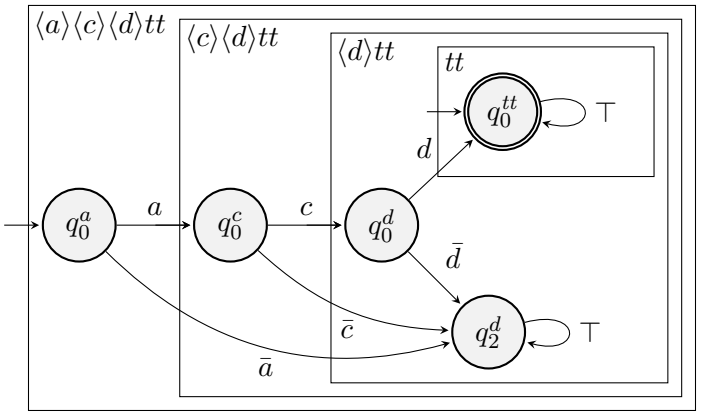


Figure 5.7. The automaton for $\langle a\rangle\langle c\rangle\langle d\rangle tt$.

Finally, we compute the automaton of φ' by taking the union of the automata of the two subformulas, i.e. $\mathcal{A}_{\varphi'} = \mathcal{A}_{\langle a\rangle\langle c\rangle\langle d\rangle tt} \cup \mathcal{A}_{\langle b\rangle\langle c\rangle\langle d\rangle tt}$ (Figure 5.8):

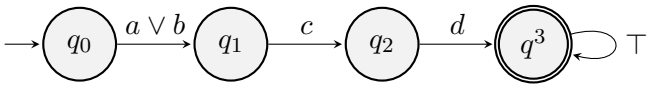


Figure 5.8. The automaton $\mathcal{A}_{\varphi'} = \mathcal{A}_{\langle a\rangle\langle c\rangle\langle d\rangle tt} \cup \mathcal{A}_{\langle b\rangle\langle c\rangle\langle d\rangle tt}$.

Example 5.3. *This example shows how the translation works for $\langle\rho\rangle\psi$ when ρ is test-free. Let $\varphi = [a^*]\langle b\rangle tt$. As before, we start from the deepest subformula in the syntax tree, i.e. tt, and then go backward. Therefore we start from $\mathcal{A}_{tt}$ showed in Figure 5.9:*

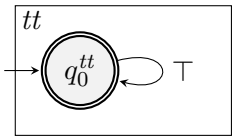


Figure 5.9. The automaton for tt.

Next, we process $\langle b\rangle tt$, yielding $\mathcal{A}_{\langle b\rangle tt}$ (Figure 5.10):

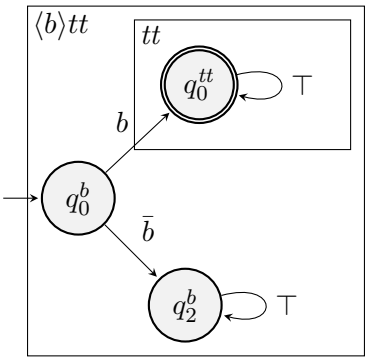


Figure 5.10. The automaton for $\langle b\rangle tt$.

Now, we consider $[a^*]\langle b\rangle tt$. The translation rules tell us to first convert the formula into $\neg\langle a^*\rangle\neg(\langle b\rangle tt)$. We first process the innermost negation, $\neg(\langle b\rangle tt)$, or equivalently, $[b]ff$. In terms of automata operations, this translates into complementing $\mathcal{A}_{\langle b\rangle tt}$, i.e. $\mathcal{A}_{[b]ff} = \overline{\mathcal{A}_{\langle b\rangle tt}}$ (Figure 5.11):

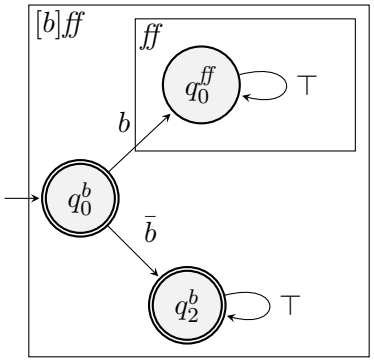


Figure 5.11. The automaton for $\neg\langle b\rangle tt$, or equivalently $[b]ff$, is obtained by complementing $\mathcal{A}_{\langle b\rangle tt}$: $\mathcal{A}_{[b]ff} = \overline{\mathcal{A}_{\langle b\rangle tt}}$.

Note that the automaton depicted in Figure 5.11 is equivalent to the formula $[b]ff$.

The next step in order to process $\langle a^*\rangle\varphi$ is to first compute the automaton for $\langle a\rangle end$ (the automaton on the left in Figure 5.12).

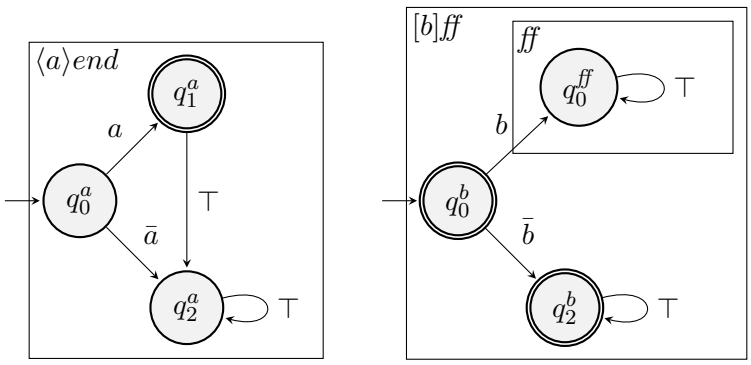


Figure 5.12. The automata $\mathcal{A}_{\langle a\rangle end}$ and $\mathcal{A}_{[b]ff}$, respectively.

Next, we compute the Kleene closure of $\mathcal{A}_{\langle a\rangle end}$ (the automaton on the left in Figure 5.13):

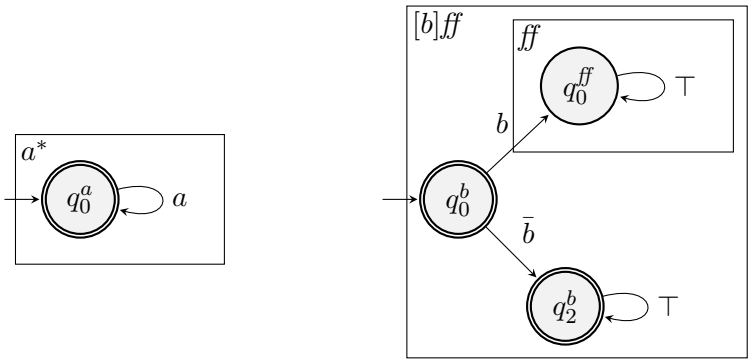


Figure 5.13. The automata $\mathcal{A}_{\langle a^*\rangle end}$ and $\mathcal{A}_{[b]ff}$, respectively.

Now, we need to compute the concatenation between the two automata, i.e. we add the ϵ-transition from the accepting states of the first operand to the initial state

of the second operand (see Section 2.5 for details), which yields the ϵ-NFA $\mathcal{A}^{\epsilon}_{N}$ in Figure 5.14:

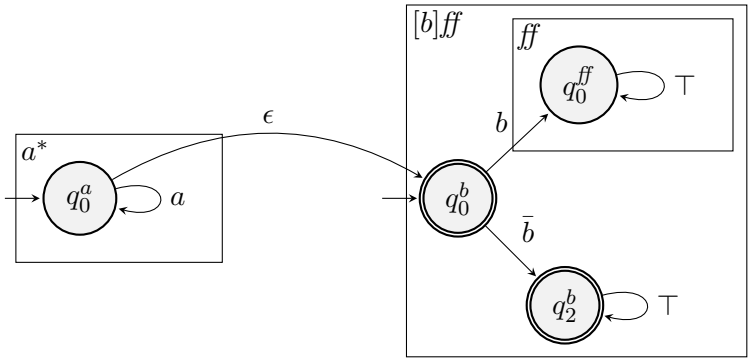


Figure 5.14. The ϵ-NFA that represents the concatenation between $\mathcal{A}_{\langle a^* \rangle end}$ and $\mathcal{A}_{[b]ff}$.

The next step is to determinize $\mathcal{A}^{\epsilon}_{N}$, yielding $\mathcal{A}_{\langle a^* \rangle [b]ff}$ (Figure 5.15):

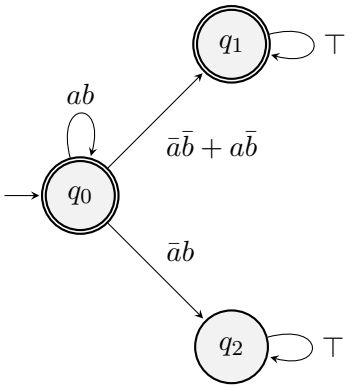


Figure 5.15. The DFA $\mathcal{A}_{\langle a^* \rangle [b]ff}$.

Since our original formula was $[a^*]\langle b \rangle tt$, we have to complement the automaton $\mathcal{A}_{\langle a^* \rangle [b]ff}$ in order to obtain the desired result $\mathcal{A}_{[a^*]\langle b \rangle tt}$, i.e. $\mathcal{A}_{[a^*]\langle b \rangle tt} = \overline{\mathcal{A}_{\langle a^* \rangle [b]ff}}$ (Figure 5.16):

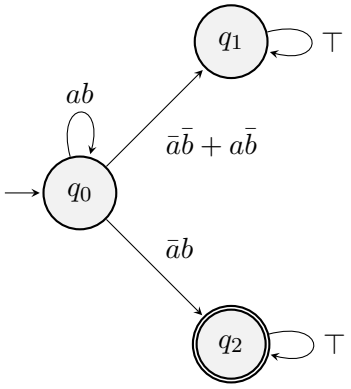


Figure 5.16. The DFA $\mathcal{A}_{\langle a^* \rangle [b]ff}$.

Example 5.4. *In this example, we will see how the technique works for formulas of the type $\langle\rho^*\rangle\psi$ when ρ is not test-free. Let $\varphi = \langle(\langle a;a\rangle tt?;true)^*\rangle\langle b\rangle tt$. The formula has a test expression $\langle a;a\rangle tt$, which is non-atomic, i.e. it requires at least two steps to be verified.*

The first step is to precompute automata $\mathcal{A}_{\langle a;a\rangle tt}$ and $\mathcal{A}_{\langle b\rangle tt}$, as prescribed by the translation rules (Figure 5.17):

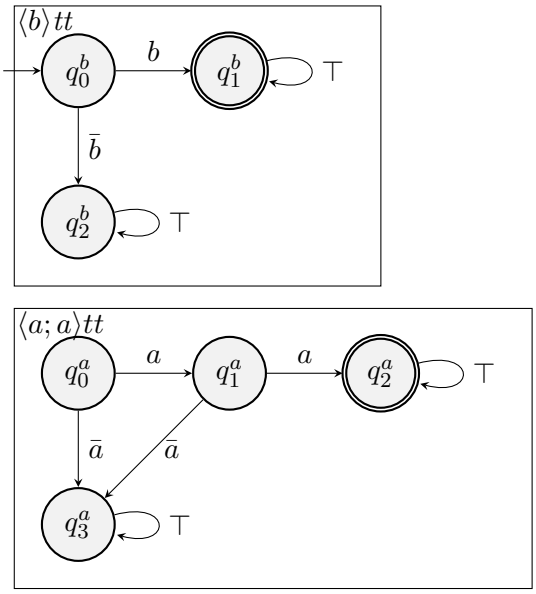


Figure 5.17. The automata $\mathcal{A}_{\langle a,a\rangle tt}$ and $\mathcal{A}_{\langle b\rangle tt}$, respectively.

Since ρ contains test expressions, we simply fall back to using the classical algorithm that computes the AFA *from $\langle\rho^*\rangle\psi$, $\mathcal{A}_A$. We start from the initial state $q_0 = \varphi$. We expand q_0 using the function ∂:*

$$\begin{aligned}\partial(\varphi,\Pi) &= \partial(\text{“}\langle b\rangle tt\text{”},\Pi) \vee (\partial(\text{“}\langle a;a\rangle tt\text{”},\Pi) \wedge \partial(\text{“}\boldsymbol{E}(\langle true\rangle \boldsymbol{F}_\varphi\text{”}),\Pi))\\ &= \partial(\text{“}\langle b\rangle tt\text{”},\Pi) \vee (\partial(\text{“}\langle a;a\rangle tt\text{”},\Pi) \wedge \partial(\text{“}\langle true\rangle\varphi)\text{”},\Pi))\end{aligned}$$

The alternating automaton $\mathcal{A}_A$ would have the following transitions from q_0:

State	$\delta_A(q_0,\{a,b\})$	$\delta_A(q_0,\{a\})$	$\delta_A(q_0,\{b\})$	$\delta_A(q_0,\{\})$
q_0	$q_1^b \vee (q_1^a \wedge q_0)$	$q_2^b \vee (q_1^a \wedge q_0)$	$q_1^b \vee (q_3^a \wedge q_0)$	$q_2^b \vee (q_3^a \wedge q_0)$

Intuitively, we reinterpret states of $\mathcal{A}_{\langle a,a\rangle tt}$ and $\mathcal{A}_{\langle b\rangle tt}$ as if they were states of alternating automata. The concatenating transitions in Table 5.4 start from q_0 and lead to q_0 or states of the two automata in Figure 5.17, according to the read symbol and to the alternation specified by $\partial(\varphi,\Pi)$. Also, note that some transitions lead to either acceptance (e.g. $\delta_A(q_a,\{a,b\}) = true$, since q_1^b is an accepting sink state) or rejection (e.g. $\delta_A(q_a,\{\}) = false$, since q_3^a is a rejecting sink state).

The technique guarantees us that the automaton $\mathcal{A}_A$ is semantically equivalent to φ. The next step is to determinize $\mathcal{A}_A$ to obtain a DFA*. In Figure 5.18 you can see the determinized $\mathcal{A}_A$, which corresponds to $\mathcal{A}_{\langle(\langle a;a\rangle tt?;true)^*\rangle\langle b\rangle tt}$.*

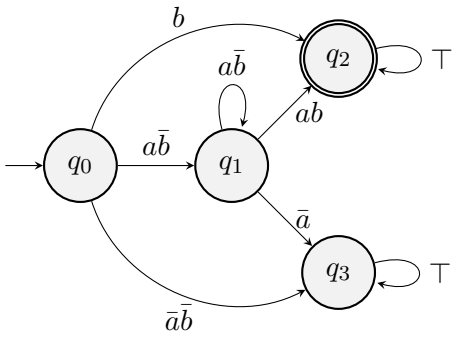


Figure 5.18. The automaton $\mathcal{A}_{\langle(\langle a;a\rangle tt?;true)^*\rangle\langle b\rangle tt}$.

5.4 Summary and Discussion

This chapter introduced one of the main theoretical contribution of the thesis: a compositional approach to transform LTL$_f$/LDL$_f$ into DFA. Despite there were already several transformation approaches in the literature, they do not rely on a direct translation from LTL$_f$/LDL$_f$ formulas into DFA, but go through alternating automata or other logic formalisms like first-order logic. We formalized the approach using a structural induction scheme that associates an automata operation with each formula operation, starting from elementary automata. We proved the correctness of the technique and analyzed its computational complexity, which we assessed to be non-elementary in the size of the formula. Although it might sound scary, we observe that the procedure implemented by a state of the art tool `Mona` is non-elementary too, and nevertheless it works surprisingly well in practice. The implementation and the experimental evalutation of our approach will be the topic of the next two chapters. In this chapter, we also provided several examples of the technique at work.

We now discuss specific topics regarding the novel approach, and foresee future research directions.

5.4.1 Refinement of Complexity Analysis

The compositional approach presented in this chapter has been shown to have complexity which is non-elementary wrt the size of the formula, due to an arbitrary number of nested exponential operations. Nevertheless, as we will see in Chapter 7, an careful implementation exploiting efficient semi-symbolic data structures can overcome this theoretical limitation, and in fact perform better than other approaches in practice.

This highlights a recurring topic in classical computational complexity, which is famous to be exclusively focused on worst-case analysis. Apparently, most of the LDL$_f$ formulas considered in the experiments do not fall in the category of formulas where the blow-up in state space of the minimal equivalent DFA is doubly-exponential.

As a future work, would be interesting to outline a taxonomy of LDL$_f$ formulas for which the worst-case complexity analysis of the compositional approach can be improved.

Another idea is to consider a different criterion of complexity, e.g. the average-case criterion (Bogdanov and Trevisan, 2006). Given a probability distribution over inputs (i.e. over LDL$_f$ formulas), we can pursue a formal analysis on how the most expensive operations (projections, concatenations, closures etc.) are likely to occur

in practice, and therefore give the right weight to such instances when averaging over all possible inputs in the analysis. The analysis can then be verified experimentally by sampling from the distribution over formulas and measure the running time of the implementation. This is motivated by similar observations in the literature regarding automata determinization, where it is argued that in practice the state space size of the DFA is not going to explode and remains tractable (Tabakov and Vardi, 2005; Basin and Klarlund, 1996; Basin and Klarlund, 1995), especially when considering the minimized version of the DFA (Zhu, Tabajara, Pu, et al., 2021).

5.4.2 Tailored Rewriting of LDL$_f$ Formulas

Many systems in the area of software verification where the manipulation of formulas is involved do apply some simplfication and rewritings of the input formula in order to make it suitable for the specific procedures that process it (e.g. see (Duret-Lutz et al., 2016) or (Henriksen, L. Jensen, et al., 1995)). In our case, we can devise rewriting rules that preserve the semantic meaning of the formulas but their output gives some computational advantages from the perspective of the compositional approach, e.g. whenever it is possible, remove a costly operation that implies a determinization step.

5.4.3 Design Compositional Translation for Other Formalisms

The compositional approach transforms an LDL$_f$ formula into a DFA. In order to translate an LTL$_f$ formula using a compositional approach, one has to first translate the LTL$_f$ formula into an LDL$_f$ formula, and then apply the same technique as before. However, one could think of a direct translation from LTL$_f$ into DFAs, using an analogous approach which focuses on compositionality. Intuitively, since LTL$_f$ is strictly less expressive than LDL$_f$, we can conjecture that using direct transformation rules from LTL$_f$ would give practical advantages (despite the complexity being the same).

Analogously, we can extend the approach to the pure-past versions of LTL$_f$ and LDL$_f$, namely PPLTL and PPLDL (De Giacomo, Di Stasio, et al., 2020). From a computational complexity perspective, reasoning with such formalisms is easier than reasoning with the future fragments LTL$_f$/LDL$_f$. Would be interesting to understand whether the compositional approach applied to such formalisms could give even greater performances.

Chapter 6

Symoblic Compositional Approach

In this chapter, we give a more concrete formalization of the compositional approach presented in Chapter 5, important for efficient implementations of the technique, as we shall see in Chapter 7.

The chapter is structured as follows:

- In Section 6.1, we introduce a novel technique to represent alternating finite automata by means of deterministic finite automata with an extended alphabet to represent the alternating transitions, showing its correctness wrt the semantics.
- In Section 6.2, we describe how we use semi-symbolic automata operations like concatenation and Kleene closure for the compositional approach, and how we exploit the novel transformation introduced in Section 6.1 to handle the case $\langle \rho^* \rangle \varphi$ with ρ non-test-free.
- Section 6.3 concludes the chapter and proposes future research directions.

The contents of this chapter have been partially published in the conference paper (De Giacomo and Favorito, 2021).

6.1 From AFA to DFA using projections

In this section, we describe a novel technique to determinize an AFA in semi-symbolic representation in order to obtain a DFA. This will be crucial in the implementation in handling the case $\langle \rho^* \rangle \psi$ when ρ contains tests, as a naive implementation would build the entire AFA of the formula and then go through the double subset construction to first get an NFA and then a DFA, whereas in our case, as we will see, the new technique allows minimizing more often while in the determinization process.

Let $\mathcal{A}_A$ be an AFA, and assume it is in semi-symbolic representation, i.e. its alphabet is $\Sigma = 2^{\mathcal{P}}$ for some set of bits $\mathcal{P}$. Our technique relies on building intermediate DFAs, similarly to what has been done for concatenation and Kleene closure (see Section 2.6), on a bigger alphabet with only additional existential bits. Since we need to model universal choice in alternating transitions, we will have two types of auxiliary bits: *existential* bits, which model the existential choice, to be projected via EPROJECT(as before) and *universal*, which model universal

choice, to be projected via UPROJECT. Let q be the current state to expand in the computation of the AFA, and let ϕ_q the formula over Q that determines the next transitions. Without loss of generality, assume ϕ_q is in disjunctive normal form, and assume that each clause is indexed across all the products and each atom occurrence is indexed within its clause. Let us call such indices i and j, respectively. We start the construction of the DFA iteratively, adding states and transitions from the AFA. For each outgoing transition from the AFA state q, the construction adds a DFA state q' and an outgoing transition for each atom occurrence (i.e. an AFA state), whose guard is determined by the guard of the current transition in conjunction with the instantiation of the existential and universal bits, corresponding to the binary representation of the indices i and j, respectively. Intuitively, to obtain the DFA corresponding to the AFA, instead of doing the subset construction on-the-fly (which would need to keep track of *sets of sets* of states), we push the representation of the alternation in the alphabet, through the addition of universal and existential bits. This exploits the asymmetry of the semi-symbolic representation, which is symbolic in the transitions and explicit in the states. Hence, it is less costly, in terms of required space, to add a transition rather than a state. Moreover, this also gives the opportunity to minimize the resulting DFA, hence saving computational resources for the following projections and determinizations.

More formally, let $\mathcal{A}_A = \langle Q_A, 2^{\mathcal{P}}, q_0, F_A, \delta_A \rangle$ be a AFA in semi-symbolic representation over the set of atomic propositions $\mathcal{P}$. Assume without loss of generality that for every $q \in Q$, and $\alpha \in 2^{\mathcal{P}}$, $\delta_A(q, \alpha)$ is in disjunctive normal form (DNF), and that it is a positive boolean formula (i.e. no negations in front of literals). Before constructing the intermediate DFA, we need to introduce preliminary concepts and notations. Let ϕ being a positive propositional formula in DNF; we define:

- A *literal* of ϕ is an atomic proposition in ϕ;
- A *clause* is an ordered list of literals; we need the notion of ordering because we will index the literals within a clause.
- $literals(\phi)$: all the literals that occur in ϕ;
- $|\phi_i|$: the number of literals that occur in the i-th clause ϕ_i;
- ϕ_{ij}: the j-th literal in the i-th clause ϕ_i, with $1 \leq j \leq |\phi_i|$. We assume there is an ordering defined over literals, which induces the indexing;
- $clauses(\phi)$: the ordered list of clauses in ϕ; we need the notion of ordering because we will index the clauses of ϕ; we assume a lexicographic order over clauses induced by the ordering over literals;
- $|\phi| = |clauses(\phi)|$: the number of clauses in ϕ;
- ϕ_i: the i-th clause in ϕ, with $1 \leq i \leq |\phi|$;

Let $n = \max_{q,\alpha}\{|\phi'| \mid \delta(q, \alpha) = \phi'\}$ the maximum number of clauses across all formulas in δ, and let $m = \max_{q,\alpha}\{|\phi_i| \mid \delta(q, \alpha) = \phi, 1 \leq i \leq |\phi|\}$ the maximum number of literals within a clause of ϕ. Let also $N = \lceil \log_2 n \rceil$ and $M = \lceil \log_2 m \rceil$. Consider the extended set of bits $\mathcal{P}' = \mathcal{P} \cup \{e_1, \ldots, e_N\} \cup \{u_1, \ldots, u_M\}$, where e and u stands for existential and universal bits, respectively. Let $bin_B : \mathbb{N} \to \{0, 1\}^B$ be the function that transforms an integer number to its binary representation using B bits (for simplicity, if the number of bits is not enough to represent the integer, then the function is not defined). Given a (positive) propositional formula ϕ, consider the extended version of ϕ, called $\tilde{\phi}$, defined as:

$$\tilde{\phi} = \bigvee_{i=1}^{|\phi|} \left(\bigwedge_{j=1}^{|\phi_i|} \phi_{ij} \wedge \bigwedge_{j=|\phi_i|+1}^{2^M} true \right) \vee \bigvee_{i=|\phi|+1}^{2^N} false \tag{6.1}$$

The idea is to add trivial *true* literals to every clause in conjunction with the other literals in such a way that all the clauses have the same number of literals 2^M, and to add trivial $false$ clauses so to get 2^N clauses. Note that $\tilde{\phi}$ is still in DNF and it is equivalent to ϕ. Let $\tilde{\delta}(q,\alpha) = \widetilde{\delta_A(q,\alpha)}$, i.e. apply the extension operation to every $\delta_A(q,\alpha)$. Note that in this case the atoms of the extended formulas are states of the automaton, i.e. $\phi_{ij} \in Q_A$.

We build an intermediate DFA $\mathcal{A}' = \langle Q', 2^{\mathcal{P}'}, q'_0, F', \delta' \rangle$ as follows:

- $Q' = Q_A$
- $q'_0 = q_{A,0}$
- $F' = F_A$
- $\delta'(q, \alpha e_1 \cdots e_N u_1 \cdots u_M) = \tilde{\delta}(q,\alpha)_{i,j}$, where $bin_N(i) = e_1 \cdots e_N$ and $bin_M(j) = u_1 \cdots u_M$.

Intuitively, the *deterministic* function δ' encodes $\tilde{\delta}$ by pushing the alternation into the binary alphabet. A particular instantiation of the existential bits $e_1 \cdots e_N$ and the universal bits $u_1 \cdots u_M$ represents one particular step over *some* run tree starting from q after reading the symbol α. Interestingly, it is easy to see that there is a one-to-one correspondence between run trees of $\mathcal{A}_A$ and words extended with existential bits.

The following theorem proves the correctness of the technique when we (universally) project together all the universal bits, and then we (existentially) project all the existential bits:

Theorem 6.1. *Let $\mathcal{A}_A = \langle Q_A, 2^{\mathcal{P}}, q_{A,0}, F_A, \delta_A \rangle$ be an alternating automaton in semi-symbolic representation. Let $\mathcal{A}' = \langle Q', 2^{\mathcal{P}'}, q'_0, F', \delta' \rangle$ be the* DFA *as defined above. Let $\mathcal{A} = EProject(UProject(\mathcal{A}', \{1,\ldots,M\}), \{1,\ldots,N\})$. Then, $\mathcal{L}(\mathcal{A}_A) = \mathcal{L}(\mathcal{A})$.*

Proof. The claim can be rewritten as:

$$\forall w : w \in \mathcal{L}(\mathcal{A}_A) \iff w \in \mathcal{L}(\mathcal{A})$$

Note that if δ_A is deterministic, i.e. $|\delta_A(q,\alpha)| = 1$ and $|\delta_A(q,\alpha)_1| = 1$ for all q and α, then $n = 1$ and $m = 1$, which means $N = \lceil \log_2 1 \rceil = 0$ and $M = \lceil \log_2 1 \rceil = 0$, and the claim is trivially true. We prove this by induction on the length of the word w.

- $w = \epsilon$. We have that (i) $w \in \mathcal{L}(\mathcal{A}_A)$ iff $q_{A,0} \in F_A$ and (ii) $w \in \mathcal{L}(\mathcal{A})$ iff $\{\{q_0\}\} \in F$ both by construction, but since $F_A = F'$ and $q_0 \in F'$ iff $\{\{q_0\}\} \in F$, we get the thesis.
- Assume the claim holds for a generic w of length k, and we want to prove the claim for $w' = w\alpha$, with $\alpha \in 2^{\mathcal{P}}$. Let $q \in 2^{2^{Q_A}}$ be the state resulting from the run of $\mathcal{A}$ over w, i.e. $q = \delta^*(q_0, w)$. Let $\phi' = \delta^*_A(q_{A,0}, w')$ be the formula describing the $(k+1)$-th level of any run tree over $\mathcal{A}_A$, and let $\tilde{\phi}$ be its DNF form with literals and clauses sorted according to the lexicographic ordering over the states.

Sufficient condition: $w' \in \mathcal{L}(\mathcal{A}_A) \Rightarrow w' \in \mathcal{L}(\mathcal{A})$**.** If $w' \in \mathcal{L}(\mathcal{A}_A)$, then there exist a run tree r such that all nodes at depth $(k-1)$-th are labeled by states in F_A. Equivalently, it means that there exist a clause in $\tilde{\phi}$, say ϕ_i, such that all for all literals $\phi_{ij} \in \phi_i$, either $\phi_{ij} \in F_A$ or $\phi_{ij} = true$. Let $e_1 \cdots e_N$ be the instantiation of the existential bits such that $bin_N(i) = e_1 \cdots e_N$. By construction of $\mathcal{A}'$, we have that for all possible instantiations of universal bits $u_1 \cdots u_M$, $\delta'^*(q'_0, w\langle \alpha e_1 \cdots e_N u_1 \cdots u_M \rangle) = \tilde{\phi}_{ij}$ and $\tilde{\phi}_{ij} \in F'$, with $bin_M(j) = u_1 \cdots u_M$. Now, let $\mathcal{A}'' = \text{UPROJECT}(\mathcal{A}', \{1, \ldots, M\})$. By the language-theoretic definition of UPROJECT (see Equation 2.2) and by the argument above, we have that $w\langle \alpha e_1 \cdots e_N \rangle \in \mathcal{L}(A'')$ Let $\mathcal{A} = \text{EPROJECT}(\mathcal{A}'', \{1, \ldots, N\}$. By the language-theoretic definition of EPROJECT (see Equation 2.1), we have that $w\alpha \in \mathcal{L}(\mathcal{A})$ since there exists the word $w\langle \alpha e_1 \cdots e_N \rangle$ that belongs to the language $\mathcal{L}(\mathcal{A}'')$ being projected over all the existential bits. Therefore, the thesis holds.

Necessary condition: $w' \in \mathcal{L}(\mathcal{A}_A) \Leftarrow w' \in \mathcal{L}(\mathcal{A})$**.** If $w' \in \mathcal{L}(\mathcal{A})$, then by definition of EPROJECT there exist a word $w_e = w\langle \alpha e_1 \cdots e_N \rangle \in (2^{\mathcal{P} \cup \{e_1, \ldots e_N\}})^*$ such that $w_e \in \mathcal{L}(\mathcal{A}'')$, where $\mathcal{A}''$ is such that $\mathcal{A} = \text{EPROJECT}(\mathcal{A}'', \{1, \cdots, N\})$. Moreover, by definition of UPROJECT, for all $u_1 \cdots u_M$, we have that $w\langle \alpha e_1 \cdots e_N u_1 \cdots u_M \rangle \in \mathcal{L}(\mathcal{A}')$. By construction of $\mathcal{A}'$, it means that for all $\tilde{\phi}_{ij} \in \tilde{\phi}_i$, we have $\tilde{\phi}_{ij} \in F_A$, with $bin_N(i) = e_1 \cdots e_N$. But that means there exist a run for $\mathcal{A}_A$ obtained as an extension of a run from w with α that has at the $(k+1)$-th level all the $\tilde{\phi}_{ij}$ states is accepting, and therefore that $w' \in \mathcal{L}(\mathcal{A}_A)$.

□

In our implementation, we project bits one after the other so to have more opportunity to minimize the partial results. This does not compromise correctness.

In Section 6.2.3 there will be examples of this determinization technique in the context of `Lydia` implementation.

6.2 Semi-symbolic automata operations

In this section, we describe the operations involved and

6.2.1 Existential and Universal Projections

The existential projection of the ith bit ($1 \leq i \leq k$), and the determinization of its result, denoted as $\text{EPROJECT}(\mathcal{A}, i)$ converts a DFA $\mathcal{A}$ recognizing a language L to a DFA $\mathcal{A}'$ recognizing the language L' where L' is the existential projection over bit i of L. The process consists of removing the i-th track of the alphabet and determinizing the resulting non-deterministic automaton via on-the-fly subset construction. The universal projection, denoted as $\text{UPROJECT}(\mathcal{A}, i)$, is also based on the subset construction used by the existential one, however, while in the existential projection the acceptance is true iff *exists* a state in the subset that is accepting, whereas in the universal projection the acceptance is true iff *all* the states in the subset are accepting. These two operations will be important building blocks for other operations.

6.2.2 Concatenation and Kleene Closure

The technique presented in the previous section requires adding nondeterministic transitions to the DFA operands of certain operations. In the case of concatenation between two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, the nondeterministic choice is made in the accepting states of $\mathcal{A}_1$, F_1, because these states should behave as if they were the initial state of $\mathcal{A}_2$; this can be implemented by adding all the transitions that leave the initial state of $\mathcal{A}_2$, q_0^2, to the states of $\mathcal{A}_1$ in F_1. Analogously, in the case of the Kleene closure of $\mathcal{A}$, the accepting states F should additionally behave as if they were the initial state of $\mathcal{A}$, q_0, and it can be implemented by adding all the transitions that leave the accepting state q_0 to the states in F. See Section 2.6.3 for a better formalization of the concatenation operation.

In general, the new transitions might render the automaton nondeterministic. To represent the existential choice induced by the concatenation or Kleene closure, we add an auxiliary fresh bit e to the alphabet, and in the states where the nondeterministic choice happens, we use e to resolve the non-determinism, so to make the transitions deterministic. For example, in the concatenation described above, the transitions from each $f \in F_1$ that belongs to $\mathcal{A}_1$ will have the bit e set to true, whereas the new transitions will have the bit set to false, $\bar{e}$; similarly, in the Kleene closure, the old transitions will have the bit e set to true, and the new transitions will have it set to false $\bar{e}$. This will ensure that the result of those operations is still a DFA, as $e\phi_1 \wedge \bar{e}\phi_2 = \bot$, with ϕ_1 and ϕ_2 being propositional formulae. Finally, the desired automaton is $\mathcal{A} = \text{EPROJECT}(\mathcal{A}', i_e)$, where i_e is the index of the bit e. See Section 2.6.3 for a better formalization of the Kleene closure operation.

Example 6.2. *Let $\varphi = \langle a^* \rangle \langle b \rangle tt$. The translation rules specify that the first step is to compute the automaton for $\langle a \rangle end$ and $\langle b \rangle tt$ separately (Figure 6.1):*

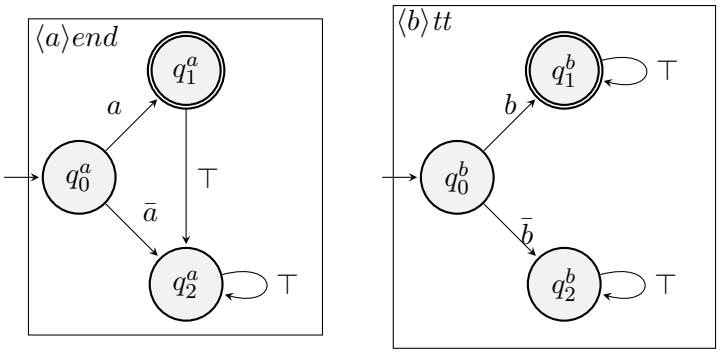


Figure 6.1. On the left the automaton $\mathcal{A}_{\langle a \rangle end}$; on the right the automaton $\mathcal{A}_{\langle b \rangle tt}$.

Now, we need to compute the Kleene closure of $\mathcal{A}_{\langle a \rangle end}$. Similarly as it has been done in Example 2.5, we proceed by (i) adding an auxiliary existential bit e; (ii) updating the current transitions with the new bit set to 0*; (iii) adding the closure transitions with the new bit set to* 1*; and (iv) making the initial state accepting:*

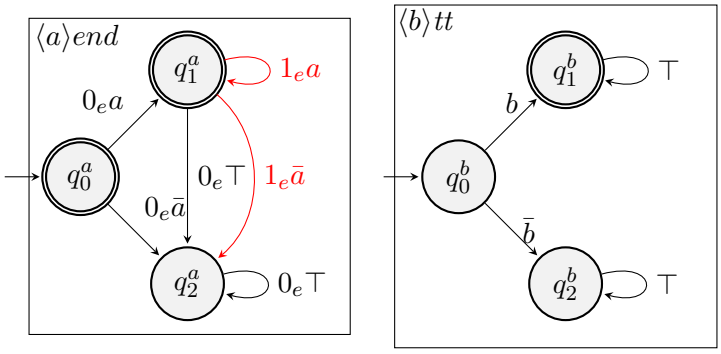


Figure 6.2. On the left the temporary automaton $\mathcal{A}'_{\langle a\rangle end}$ with the Kleene closure transitions (in red); on the right the automaton $\mathcal{A}_{\langle b\rangle tt}$.

The next step is to existentially project bit e in $\mathcal{A}'_{\langle a\rangle end}$, determinize and minimize the result (Figure 6.3):

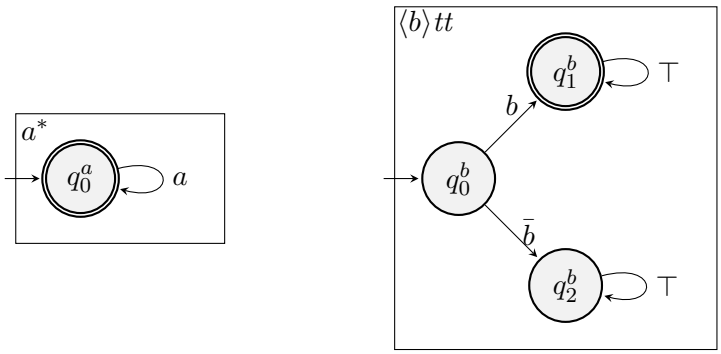


Figure 6.3. On the left the automaton $\mathcal{A}_{a^*}$, determinized and minimized; on the right the automaton $\mathcal{A}_{\langle b\rangle tt}$.

We are ready to concatenate $\mathcal{A}_{a^*}$ and $\mathcal{A}_{\langle b\rangle tt}$. First, we add the concatenation transitions (Figure 6.4):

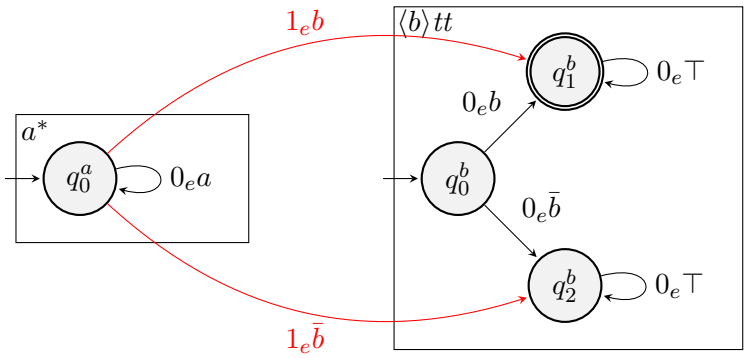


Figure 6.4. The two automata after the placement of concatenating transitions.

Then, we project the auxiliary bit e, determinize and minimize the result (Figure 6.5:

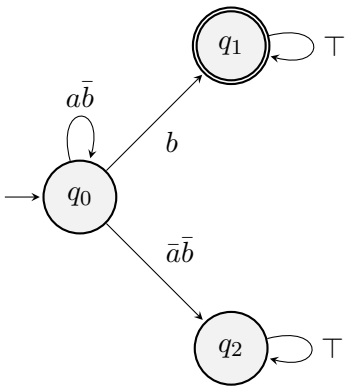


Figure 6.5. The automaton $\mathcal{A}_{\langle a^* \rangle \langle b \rangle tt}$.

6.2.3 Construction of the AFA

When we handle the case of $\langle \rho^* \rangle \varphi$ and $\rho \not\equiv \langle \rho \rangle end$, because of tests, we need to resort to constructing an AFA, and then determinizing it. Instead of considering the AFAs formalism directly, we rely on building intermediate DFAs on a bigger alphabet having two types of auxiliary bits: existential, to be projected via EPROJECT(as before) and *universal*, to be projected via UPROJECT, as explained in Section 6.1, Let q be the current state to expand in the computation of the AFA, and let ϕ_q the formula over Q that determines the next transitions. Without loss of generality, assume ϕ_q is in disjunctive normal form, and assume that each clause is indexed across all the products and each atom occurrence is indexed within its clause. Let us call such indices i and j, respectively. Then, the construction adds a transition for each atom occurrence (i.e. an AFA state), whose guard is determined by the AFA transformation rules (De Giacomo and Vardi, 2013; Brafman, De Giacomo, and Patrizi, 2018) in conjunction with the instantiation of the existential and universal bits, corresponding to the binary representation of the indices i and j, respectively.

Intuitively, to obtain the DFA corresponding to the AFA, instead of doing the subset construction on-the-fly (which would need to keep track of *sets of sets* of states), we push the representation of the alternation in the alphabet, through the addition of universal and existential bits. This exploits the asymmetry of the `Mona` DFA implementation, which is symbolic in the transitions and explicit in the states. Hence, it is less costly to add a transition rather than a state. Moreover, this also gives the opportunity to minimize the resulting DFA, hence saving computational resources for the following projections and determinizations.

A crucial difference with respect to the classic LDL$_f$-to-AFA transformation is that, whenever one of the atom occurrences we come across is either a test expression $\psi?$ or φ, instead of expanding those nodes as if they were states of the AFA, we concatenate the current state to their DFAs. This operation can be seen as a casting a DFA into a "deterministic AFA", followed by a concatenation between the current AFA under construction and the precomputed automata of the subformulas. This gives a good amount of compositionality also to this case, which translates into more opportunity to minimize the partial results, and hence in achieving greater performances.

Symbolic alternating transitions

A fundamental difference is that, differently from what we have done in Example 5.4, we do not consider all possible propositional interpretations $\Pi \in 2^{\mathcal{P}}$

(see Table 5.4), as it would not be scalable in the number of propositional symbols. Instead, whenever we need to compute the outgoing transitions from the initial states of some automaton $\mathcal{A}_{\psi?}$ or $\mathcal{A}_{\varphi}$, we exploit the `Mona` DFA representation by enumerating all the paths from the BDD root node, corresponding to the transition function starting from the automaton's initial state, to a leaf node, which identifies a successor state (see Section 7.1.2). Intuitively, each path encodes a set of propositional interpretations that would make the automaton transition leading to the same successor state.

Therefore, in order to compute the successors of an AFA state "φ", we cannot rely on the AFA transition function ∂, as defined in Section 4.1.1, because its evaluation requires a specific propositional interpretation Π. Hence, we define a variant of ∂ that does not require Π, that we call $\hat{\partial}$, defined as ∂ except the cases in which Π is used in its evaluation, i.e.:

$$\begin{aligned} \hat{\partial}(\text{“}\langle\phi\rangle\varphi\text{”}) &= \text{“}\langle\phi\rangle \boldsymbol{E}(\varphi)\text{”} \\ \hat{\partial}(\text{“}[\phi]\varphi\text{”}) &= \text{“}[\phi]\boldsymbol{E}(\varphi)\text{”} \end{aligned}$$

Example 6.3. *Let $\varphi = \langle(\langle a;a\rangle tt?;true)^*\rangle\langle b\rangle tt$, as in Example 5.4. The first step is to precompute automata $\mathcal{A}_{\langle a;a\rangle tt}$ and $\mathcal{A}_{\langle b\rangle tt}$ (Figure 5.17). Then, we start building the* AFA*, with φ as initial state. We expand q_0 using the function ∂:*

$$\begin{aligned} \hat{\partial}(\varphi) &= \hat{\partial}(\text{“}\langle b\rangle tt\text{”}) \vee (\hat{\partial}(\text{“}\langle a;a\rangle tt\text{”}) \wedge \hat{\partial}(\text{“}\langle true\rangle \boldsymbol{F}_{\varphi}\text{”})) \\ &= \text{“}\langle b\rangle tt\text{”} \vee (\text{“}\langle a;a\rangle tt\text{”} \wedge \text{“}\langle true\rangle\varphi\text{”}) \end{aligned}$$

Since the states in $\hat{\partial}(\varphi)$ are all known, the exploration of the AFA *$\mathcal{A}_{A,\varphi}$ is complete. We now want to determinize $\mathcal{A}_{A,\varphi}$ using the technique presented in Section 6.1. We start by normalizing $\hat{\partial}(\psi)$ for all ψ. The only non-atomic transition formula is precisely $\hat{\partial}(\varphi)$ since for the other states $\psi \neq \varphi$, $\hat{\partial}(\psi)$ is deterministic. We have that $n = 2$ since we have at most two clauses, and $m = 2$ since the maximum clause size is 2. This means we need only $N = 1$ existential bits and $M = 1$ universal bits.*

We associate each instantiation of e and u to every literal of $\widetilde{\hat{\partial}(\varphi)}$ according to their clause index and their position within the clause:

$$\widetilde{\hat{\partial}(\varphi)} = (\underset{e\bar{u}}{\text{“}\langle b\rangle tt\text{”}} \wedge \underset{eu}{true}) \vee (\underset{\bar{e}\bar{u}}{\text{“}\langle a;a\rangle tt\text{”}} \wedge \underset{\bar{e}u}{\text{“}\langle true\rangle\varphi\text{”}}) \tag{6.2}$$

In other words, from $q_0 = \varphi$ (the current state in this iteration):

- $\bar{e}\bar{u}$*: take* all *transitions from initial state of $\mathcal{A}_{\langle a;a\rangle tt}$;*
- $\bar{e}u$*: go to $\varphi = q_0$ (a self-loop)*
- $e\bar{u}$*: take* all *transitions from initial state of $\mathcal{A}_{\langle b\rangle tt}$;*
- eu*: go to accepting sink;*

Figure 6.6 depicts the AFA *represented as a semi-symbolic* DFA *with the auxiliary bits.*

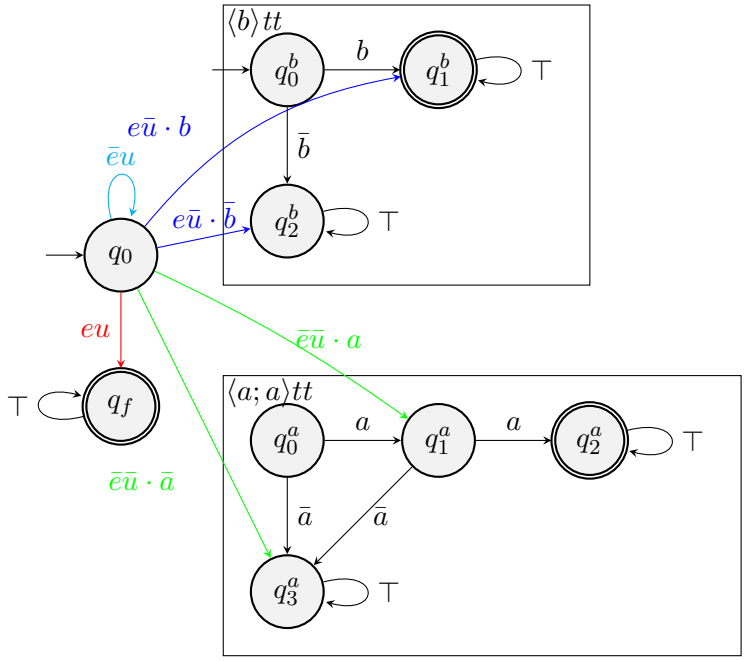


Figure 6.6. The AFA of φ represented as a DFA with additional existential and universal bits e and u. The transitions that encode the alternation are colored according to Equation 6.2.

The next steps are the universal projection of bit u and the existential projection of bit e, in that order. Figure 6.7 shows the result of the minimization of the DFA *in Figure 6.6:*

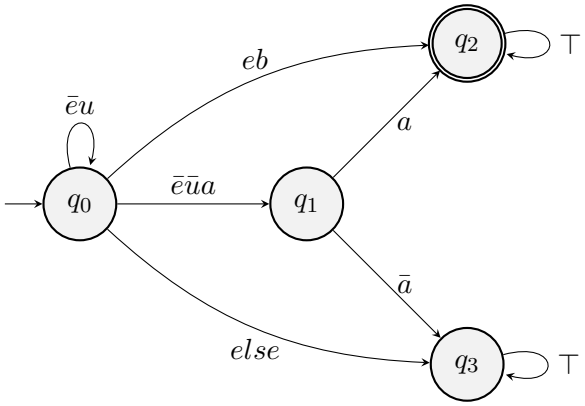


Figure 6.7. The DFA of Figure 6.6, minimized

Then, it follows the universal projection (Figure 6.8):

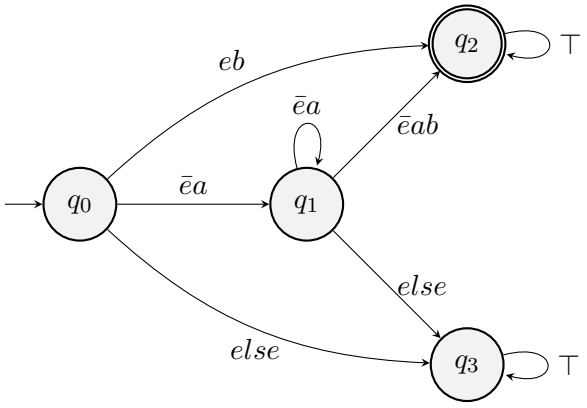


Figure 6.8. The DFA after the universal projection of bit u.

Then, the existential projection (Figure 6.9), including a final minimization step:

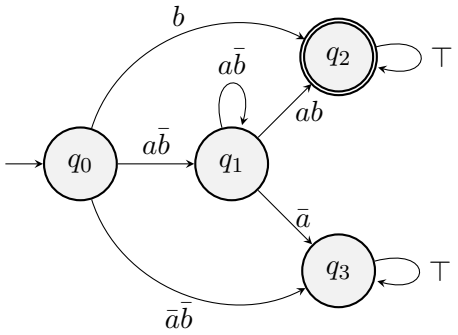


Figure 6.9. The final minimized DFA $\mathcal{A}_\varphi$.

6.3 Summary and Discussion

This chapter explained a more practical formalization of the compositional approach introduced in Chapter 5. In particular, it is a formalization that it is suitable for an efficient implementation, as we shall see in Chapter 7.

We first proposed a general technique to represent an alternating automaton by means of a deterministic finite automaton over a larger alphabet with auxiliary bits to represent the existential and the universal choices. Then, we saw how we can use the semi-symbolic DFA representations and operations for our compositional approach, and how to apply the DFA-representation of the AFA. We discussed some examples of the technique.

We now outline potential future works.

6.3.1 Exploit DFA-representation of AFAs for other problems

The DFA-representation of the AFA which pushes the alternation representation in the alphabet, as presented in Section 6.1, is very general and not tight to the compositional approach. We wonder whether such technique can be beneficial for other use-cases where the AFA is involved, especially because the DFA-representation makes it easy to reduce the state space of a DFA as there are efficient algorithms for it (e.g. the Hopcroft's minimization algorithm, which is $\mathcal{O}(n \log n)$).

6.3.2 DFA-representation of a Full AFA

Instead of relying on the DFA-representation only for the case $\langle \rho^* \rangle \varphi$ with ρ non test-free, one could devise the following translation algorithm:

1. given an $\text{LTL}_f/\text{LDL}_f$ formula φ, compute the corresponding AFA $\mathcal{A}_A$.
2. from the $\mathcal{A}_A$, compute the DFA-representation of it, $\mathcal{A}'$.
3. Determinize $\mathcal{A}'$ by doing the projections EPROJECT and UPROJECT.

Step 1 can be done in linear time to build the "structure" of the $\mathcal{A}_A$, but the normalization of the transition function δ_A might require an exponential increase in the size of the formula. We could consider a smarter approach that, considering all values $\delta_A(q, a)$, computes whether it is better to normalize using DNF or CNF for normalization and, in the former case, reverse the order of projections (first existential, then universal bits). Or, consider a smarter normalization approach that avoids the exponential blow-up, e.g. analogous to the Tseytin encoding (Tseitin, 1983) for achieving an equisatisfiable encoding of the boolean formula.

Step 3 can be carried out in different ways, each one yielding a variant of the overall algorithm which, depending on the problem being solved, can be more or less suited. In particular, we can take one of the following alternatives:

- first project all the universal bits, do the subset construction to determinize, minimize, then project all the existential bits, determinize and minimize.
- project one auxiliary bit at a time, determinize and minimize after each projection.

We could also devise a smarter subset construction that determinizes and minimizes starting from the subgraphs of the automaton The intuition is that we can decompose the determinization and minimization step by considering portions of the automaton that do not depend on the past.

Finally, would be interesting to draw inspiration and to find the relationships with the symbolic implementation of AFA on infinite traces (Bloem et al., 2007).

6.3.3 Hybrid Compositional Approach

A semi-symbolic representation of the DFA, despite being symbolic in the alphabet, is still explicit in the representation of the state space. This makes the representation not scalable in cases which require many states. However, being fully symbolic if there is no need can worsen the performances for simpler cases due to the added complexity of the compilation of the representation.

Drawing inspiration from (Bansal et al., 2020), we could devise a fully-compositional approach that starts with a semi-symbolic representation and, if during the transformation the explicit state representation becomes a bottleneck, the partial automata are transformed in a fully-symbolic representation, and the compositional transformation continues with fully-symbolic operators between DFAs. The transition to a fully-symbolic approach might worsen the performances due to higher computational costs of the operations (e.g. minimization takes $\mathcal{O}(n^2)$), but nevertheless would be able to scale to bigger automata.

Chapter 7

The Lydia and LydiaSynt Tools

This chapter describes the Lydia tool, an efficient implementation of the compositional approach presented in Chapter 5, and the LydiaSynt tool, which relies on Lydia to build the DFA and on Syft+ (Zhu, Tabajara, Li, et al., 2017) to perform $\text{LTL}_f/\text{LDL}_f$ synthesis.

The chapter is structured as follows:

- In Section 7.1, we describe the MonaDFA Library, foundational building block of our implementation.
- In Section 7.2, we outline the software architecture of Lydia and LydiaSynt, with hints on how they work internally.
- In Section 7.3, we provide extensive experimental coverage on DFA construction times and LTL_f synthesis times.
- Section 7.4 concludes the chapter and discusses potential improvements to bring in the current implementation of the tool.

The contents of this chapter have been published in the conference paper (De Giacomo and Favorito, 2021).

7.1 Mona DFA Library

In this section, we describe the Mona DFA library, an important building block of our implementation. In Section 7.1.1 we briefly describe the Mona main features and capabilities; in Section 7.1.2 we present the Mona automata representation, which exploits a novel BDD-based data structure introduced by Mona, the Shared Multi-terminal Binary Decision Diagram (shMBDD).

7.1.1 What is Mona

Mona (Henriksen, Jensen, et al., 1995; Klarlund and Møller, 2001) is an efficient implementation of the decision procedures for WS1S (see Section 4.4.2). "Efficient" here means that the tool is fast enough to have been used in a variety of non-trivial settings. MONA translates WS1S and WS2S formulas (Thatcher and Wright, 1968; Doner, 1970) into minimum DFAs. The automata are represented by shared, multi-terminal BDDs (Binary Decision Diagrams) (Bryant, 1992). The tool at version 1.4 (Klarlund and Møller, 2001) is several orders of magitude more efficient than the first experimental versions due to techniques such as BDD representation, DAGification,

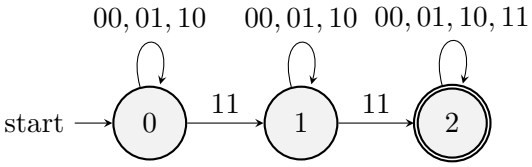


Figure 7.1. Explicit representation of an automaton accepting all strings over $\mathbb{B}^2$ with at least two occurrences of the letter 11.

and formula reductions. In addition, `Mona` provides many features, such as three-valued logics and automata, automaton visualization, importing and exporting of automata, and recursive types.

As mentioned at the beginning of this section, `Lydia` relies on the `Mona` DFA library, which provides a very space efficient data structure for automata construction and manipulation in semi-symbolic representation.

7.1.2 `Mona` automata

In `Mona`, the transitions of a DFA are symbolically represented as a shared multi-terminal binary decision diagram (Henriksen, Jensen, et al., 1995) (shMBDD), where the transition relation of a DFA is encoded as a binary decision diagram (BDD) with multiple terminal nodes. The alphabets of these DFAs are the sets of bit vectors of length k, i.e. $\mathbb{B}^k$, for some k. In our case, each bit is associated to an atomic proposition appearing in the LDL_f formula. In addition to a compact representation on transitions of DFAs, the `Mona` DFA library provides efficient implementations of standard automata operations. These operations include product, (existential) projection, determinization, and minimization. More details on how these operations are implemented can be found in the `Mona` whitepaper (Henriksen, Jensen, et al., 1995) and the user manual (Klarlund and Møller, 2001).

For example, the finite automaton accepting all strings over $\mathbb{B}^2$ with at least two occurrences of the letter 11 is shown in Figure 7.1, and its MONA representation is in Figure 7.2. The leaves of the BDD are the boxes at the bottom. Every variable is associated a unique *variable index* used for the BDD representation. The internal BDD nodes are round and contain variable indices. Each node has a *low successor* denoted by the label `lo` and a *high successor* denoted by the label `hi`.

To see how the transition table is represented, consider state 0 and the letter 10. The next state is gotten as follows: follow the pointer out of the description of state 0 in the array to the BDD node, which has index 0 corresponding to the first bit. The value of the first bit is 1, so we go to the high successor, which in turn is marked 1, denoting the second bit. Then, since this bit is 0, we take the low successor. r to the next node. That node is a leaf marked with 0. That is the value of the next state. If the second bit were 1, it would have led to the leaf node marked as 1, which would have been the new successor state.

7.2 `Lydia` and `LydiaSynt`

This section describes the high-level working of `Lydia` and `LydiaSynt`.

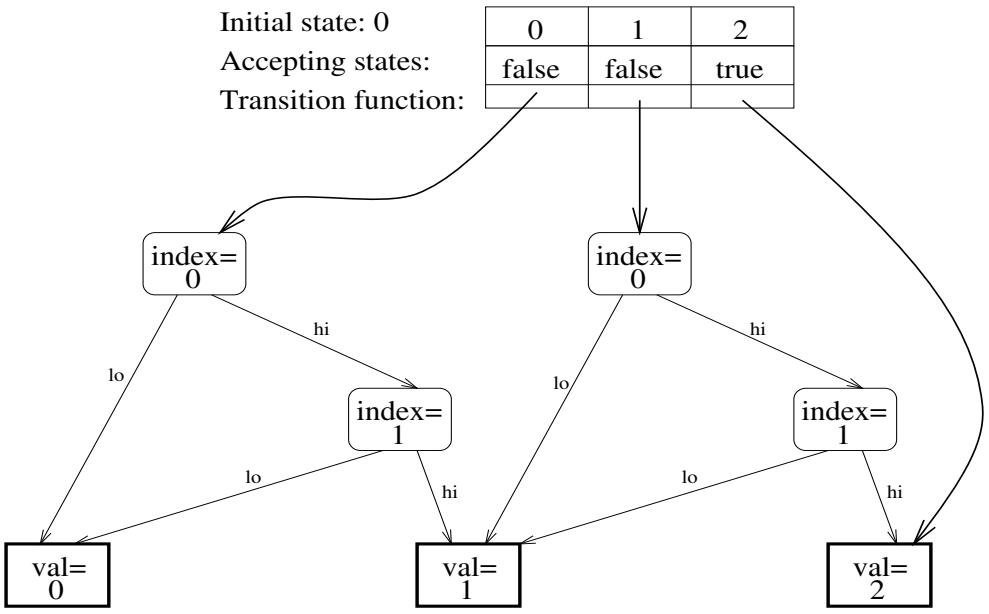


Figure 7.2. The Mona DFA equivalent to the automaton in Figure 7.2.

7.2.1 Lydia

We have implemented the compositional approach described in Chapter 5 and Chapter 6 in tool called Lydia[1], written in C++. Lydia is able to parse LTL$_f$ and LDL$_f$ using Flex and Bison (Levine, 2009) with a custom grammar[2], and represents the syntactic tree using n-ary trees. LTL$_f$ formulas are first translated in LDL$_f$ and then processed using the compositional technique. Lydia uses the Mona DFA library, described in Section 7.1 to represents DFAs and perform operations over them. Note that we do not use other Mona features related to the MSO logic parsing and manipulation. LydiaSynt is the exension of Lydia that also uses the Syft+ tool to perform LTL$_f$/LDL$_f$ synthesis, and will be described in Section 7.2.2. We extended the Mona DFA library so to include the Kleene closure, the concatenation, and the universal projection. For the Kleene closure and the concatenation, we got inspiration from the LibStranger library (Yu, Bultan, et al., 2008; Yu, Alkhalaf, and Bultan, 2010)[3]. During the induction over the formula, we adopt aggressive minimization after every step of the technique. Also, whenever the technique starts computing a product between n automata, we keep a priority queue to get the next two smallest operands; the idea is to delay state blow-up of the partial automaton as much as possible. This is a heuristics already adopted by Bansal et al. and it is crucial for better scalability.

7.2.2 LydiaSynt

LydiaSynt is the exension of Lydia that also uses the Syft+ tool to perform LTL$_f$/LDL$_f$ synthesis. After the computation of the MONA-based DFA, the Lydia tool passes it to the Syft+ tool in order to compute the winning-set.

[1] The source code of Lydia (and LydiaSynt) can be found at https://github.com/whitemech/lydia.

[2] https://marcofavorito.me/tl-grammars/

[3] https://github.com/vlab-cs-ucsb/LibStranger

`Syft`: Symbolic LTL$_f$ Synthesis

`Syft` is the implementation of a technique for doing symbolic LTL$_f$ synthesis, introduced in (Zhu, Tabajara, Li, et al., 2017). They propose a symbolic framework for LTL$_f$ synthesis based on the usual reduction to a DFA game, by following a symbolic approach based on an encoding of the DFA using boolean formulas, represented as Binary Decision Diagrams, rather than an explicit representation through the state graph. Using a symbolic approach allows them to leverage techniques for boolean synthesis (Fried, Tabajara, and Vardi, 2016) in order to compute the winning strategy. The synthesis framework employs a fixpoint computation to construct a formula that expresses the choices of outputs in each state that move the game towards an accepting state. By giving this formula as input to a boolean synthesis procedure we can obtain a winning strategy whenever one exists.

Symbolic DFA Construction.

Given an LTL$_f$ formula, `Syft` first converts the formula into a First-Order Logic formula according to the translation presented in (De Giacomo and Vardi, 2013). Then, it gives this formula as input to `Mona`, which returns a DFA $\mathcal{A}$ in semi-symbolic representation based on shared multi-terminal binary decision diagrams (shMTBDD). Then, from the ShMTBDD of the transition function δ of $\mathcal{A}$, it construct a Multi-Terminal BDD (MTBDD) for a new transition function δ', which uses a binary encoding of the states in Q using $\lceil \log_2 |Q| \rceil$ new boolean variables. Finally, it decomposes the MTBDD into a sequence of BDDs $\mathcal{B} = \langle B_0, B_1, \ldots, B_{n-1} \rangle$, with $n = \lceil \log_2 |Q| \rceil$, where each B_i, when evaluated on an interpretation of state variables and symbol variables, computes the i-th bit in the binary encoding of the successor state.

The idea of splitting the ShMTBDD into BDDs is illustrated on Figure 7.3. As shown in this example, bits b_0, b_1 are used to denote the four states s_0, s_1, s_2, s_3. In step (1), root s_0 is substituted by Z_{s_0} that corresponds to the formula $(\neg b_0 \wedge \neg b_1)$. After replacing all roots with corresponding interpretations, the MTBDD is produced. In step (2), s_0, s_1, s_2, s_3 can be represented by 00, 01, 10, 11 respectively, where b_0 denotes the leftmost bit. Bit b_0 for both s_0 and s_1 is 0. So by forcing all paths that proceed to terminals s_0 and s_1 in the MTBDD to reach terminal node 0, and all paths to terminals s_2 and s_3 to reach terminal node 1, BDD B_0 is generated. BDD B_1 is constructed in an analogous way for bit b_1.

Then, the symbolic DFA is used for running the symbolic synthesis procedure, as explained in (Zhu, Tabajara, Li, et al., 2017).

`Syft` is implemented in C++11 and uses CUDD at version 3.0.0 (Somenzi, 2015) as the BDD library. `Syft+` is an enhanced version of `Syft`, that enables dynamic variable ordering, also used by Bansal et al. `Lydia` uses `Syft+` by giving as input the `Mona` DFA directly, without passing through the `Mona` tool to translate the FOL encoding of the LTL$_f$ formula.

7.3 Experimental Evaluation

This section reports some experimental evaluation of `Lydia` and `LydiaSynt`, comparing them with other state of the art tool in LTL$_f$ synthesis.

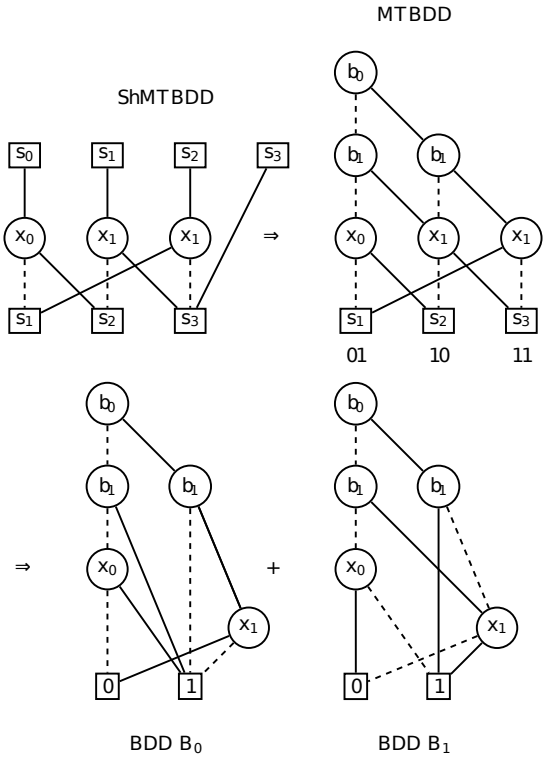


Figure 7.3. Transformation from ShMTBDD to BDD.

7.3.1 Experimental Methodology.

The evaluation has been designed to compare the performance of `Lydia` and `LydiaSynt` against their respective existing tools and approaches: `Mona` and `Lisa` for LTL$_f$-to-DFA conversion, and `Syft+` and `Lisa` for synthesis. Both LTL$_f$-to-DFA conversion tools and synthesis tools are compared on runtime and number of benchmarks solved within a given timeout.

7.3.2 Experiment Setup.

All experiments were conducted on a single laptop equipped with an Intel Core i7-8665U CPU running at 1.90GHz with 16 GB of RAM, and 300 seconds of time limit. The correctness of `Lydia` was empirically verified by comparing the results with those from all baseline tools. No inconsistencies were encountered for all solved instances.

We conduct our experiments on a benchmark suite curated from prior works, spanning classes of realistic and synthetic benchmarks: random conjunctions (400 cases) (Zhu, Tabajara, Li, et al., 2017), single counters (20 cases), double counters (10 cases) etc. and Nim games (24 cases) (Tabajara and Vardi, 2019b; Bansal et al., 2020) More details on each class can be found in the supplementary material. In the case of `Lydia`, the input LTL$_f$ formula is parsed and translated into an LDL$_f$ formula.

7.3.3 Benchmarks

We conduct our experiments on a benchmark suite curated from prior works, spanning classes of realistic and synthetic benchmarks: random conjunctions (400 cases) (Zhu, Tabajara, Li, et al., 2017), single counters (20 cases), double counters (10 cases) and Nim games (24 cases) (Tabajara and Vardi, 2019b; Bansal et al., 2020)

Random. This benchmark family has 400 instances from (De Giacomo and Favorito, 2021). The instances in this benchmark family are constructed from basic cases taken from LTL synthesis datasets Lily (Jobstmann and Bloem, 2006) and Load balancer (Ehlers, 2010). Formally, a random conjunction formula $RC(L)$ has the form: $RC(L) = \bigwedge_{1 \leq i \leq L} P_i(v_1, v_2, ..., v_k)$, where L is the number of conjuncts, or the length of the formula, and P_i is a randomly selected basic case. Variables $v_1, v_2, \ldots, v_k$ are chosen randomly from a set of m candidate variables. Given L and m (the size of the candidate variable set), we generate a formula $RC(L)$ in the following way:

1. Randomly select L basic cases;
2. For each case φ, substitute every variable v with a random new variable v' chosen from m atomic propositions. If v is an environment-variable in φ, then v' is also an environment-variable in $RC(L)$. The same applies to the agent-variables.

Single-Counter. is a simple example where the behavior of the agent is completely determined by the actions of the environment. Therefore, the challenge in this family lies mostly in proving that the specification is realizable. The agent stores an n-bit counter (where n is the scaling parameter) which it must increment upon a signal by the environment. The agent wins if the counter eventually overflows to 0. To guarantee that the game is winning for the agent, the specification assumes that the environment will send the increment signal at least once every two timesteps.

Double-Counter. is similar to the *Single-Counter* one, except that in this case there are two n-bit counters, one incremented by the environment and another by the agent. The goal of the agent is for its counter to eventually catch up with the environment's counter. To guarantee that this is achievable, the specification assumes that the environment cannot increment its counter twice in a row.

Nim. describes a generalized version of the game of Nim (Bouton, 1901) with n heaps of m tokens each. The environment and the agent take turns removing any number of tokens from one of the heaps, and the player who removes the last token loses.

7.3.4 Results and Analysis

Comparison with `Syft+`.

`Lydia` has always better runtimes than `Mona`/`Syft+`, for DFA construction and therefore for the overall synthesis running time. This suggests that working directly on $\text{LTL}_f/\text{LDL}_f$ syntax, rather than passing first through MSO or FO and then to DFA, gives better performances. This can be seen in particular for the DFA construction

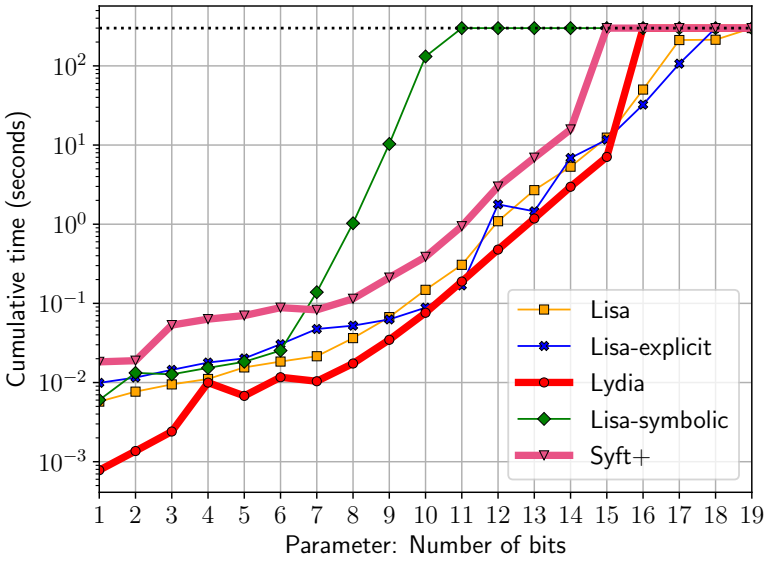


Figure 7.4. DFA construction. Runtime for single-counter benchmarks. Plots touching black line means time/memout. Timeout is at 300 sec.

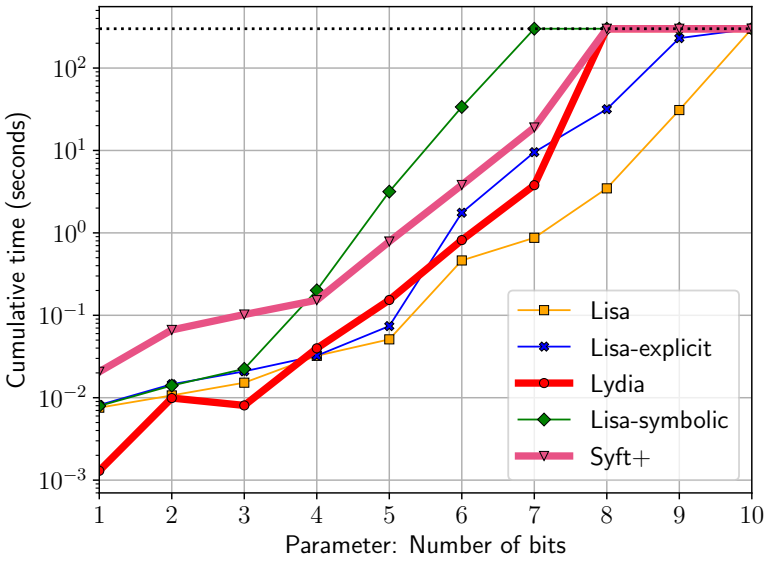


Figure 7.5. DFA construction. Runtime for double-counter benchmarks. Plots touching black line means time/memout. Timeout is at 300 sec.

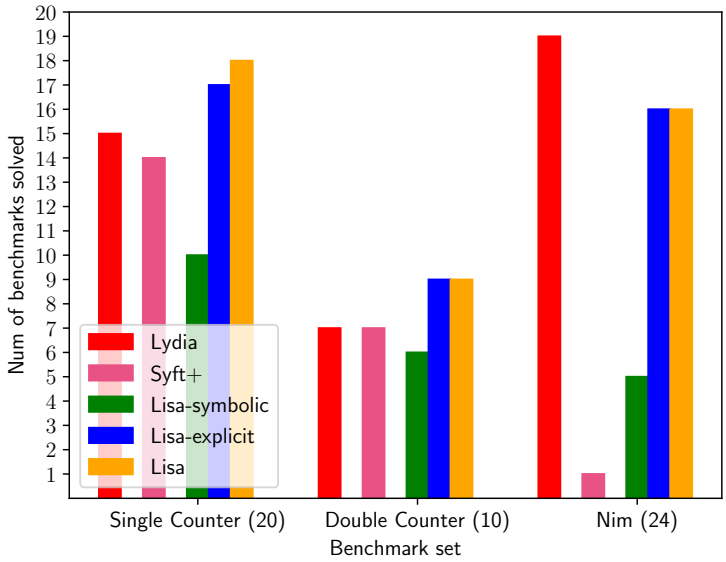


Figure 7.6. Number of benchmarks synthesized from each non-random benchmark class. Each benchmark has a timeout of 300 seconds.

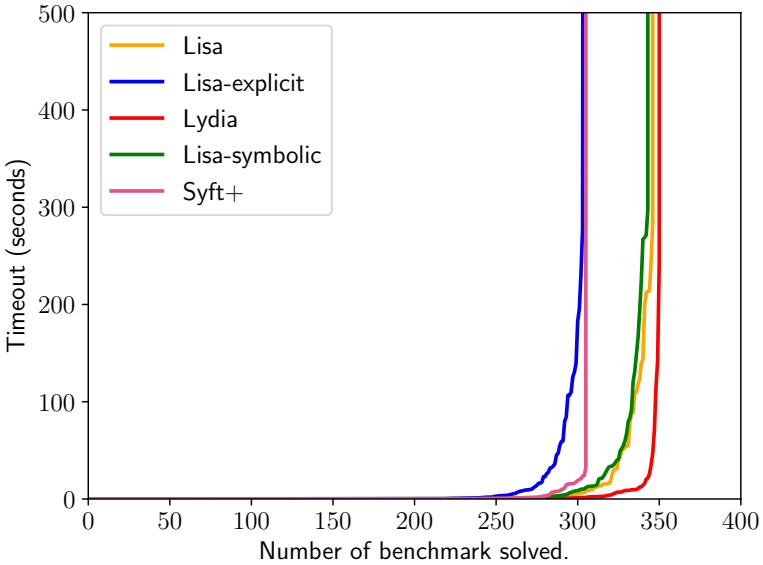


Figure 7.7. DFA construction. Cactus plot indicating number of benchmarks each tool can solve for a given timeout. Each benchmark whose running time was greater than 300 seconds was counted as ∞.

Benchmark					
Name	Lydia	Mona-based	Lisa-expl.	Lisa-symb.	Lisa
nim_1_1	**0.01**	0.15	0.07	0.07	0.07
nim_1_2	**0.02**	—	0.15	0.16	0.16
nim_1_3	**0.05**	—	0.07	1.43	0.06
nim_1_4	**0.09**	—	0.14	267.23	0.13
nim_1_5	**0.17**	—	0.27	—	0.25
nim_1_6	**0.30**	—	0.63	—	0.54
nim_1_7	**0.54**	—	1.20	—	1.02
nim_1_8	**0.82**	—	1.87	—	1.83
nim_2_1	**0.05**	—	0.14	1.49	0.10
nim_2_2	**0.20**	—	0.84	—	0.81
nim_2_3	**1.47**	—	4.95	—	4.95
nim_2_4	**7.00**	—	26.07	—	24.33
nim_2_5	**34.86**	—	125.56	—	108.86
nim_2_6	**114.87**	—	—	—	—
nim_2_7	—	—	—	—	—
nim_2_8	—	—	—	—	—
nim_3_1	**0.40**	—	3.15	—	2.67
nim_3_2	**9.93**	—	84.34	—	78.31
nim_3_3	**142.16**	—	—	—	—
nim_3_4	—	—	—	—	—
nim_4_1	**8.97**	—	110.10	—	109.79
nim_4_2	—	—	—	—	—
nim_5_1	**243.62**	—	—	—	—
nim_5_2	—	—	—	—	—

Table 7.1. Running time (in seconds) for DFA construction on the Nim benchmark set. In bold the minimum running time for a given benchmark. — means time/memout. Timeout at 300 sec.

runtime for single counter (Figure 7.4) and double counter benchmarks (Figure 7.5) and, for what concerns synthesis, in Figure 7.6, where the `Mona`-based approach, i.e. `Syft+`, is never better than `LydiaSynt`, especially on the Nim benchmark.

Comparison with `Lisa`.

We observe that `Lydia` is often better than `Lisa`. That suggests that for the explicit part of Lisa, going fully compositional is a better idea. In fact, the assumption that LTL_f formulae are conjunctions of multiple smaller subformulae might not hold in some cases, especially outside synthesis domains. This can be seen in the running times for the DFA construction on Nim benchmark (Table 7.1), the cactus plot in Figure 7.7, and in the first part of the running time of single-counter (Figure 7.4) and double-counter (Figure 7.5). However, we have to remark that for the last benchmarks of both the single and double counter, `Lisa` and `Lisa`-explicit manage to construct the DFA, whereas `Lydia` fails due to memout errors. This is due to different approaches in the computation of the DFA product: Whilst `Lydia` uses only

the `Mona` DFA library, `Lisa` relies on `Mona` for the computation of each subautomaton and then combines them with SPOT (Duret-Lutz et al., 2016). Moreover, since `Lisa` implements a hybrid approach, it is able to choose adaptively the right approach. Nevertheless, as the cactus plot in Figure 7.7 shows, `Lydia` yields better running times than `Lisa` in the majority of cases (given the timeout of 300 seconds for each benchmark). In Figure 7.6, `LydiaSynt` shows to be competitive with state-of-the-art synthesis tools like `Lisa`. However, unsurprisingly, when the problem is too large, also `Lydia` suffers from the state-space explosion, whereas `Lisa` are able to manage such inputs, thanks to their symbolic representation. Consequently, `LydiaSynt` suffers from the same limitations of `Syft+`.

A crucial thing to keep in mind is that `Lydia` processes the LTL_f formula by translating it into LDL_f and operating over it. Despite working on a more expressive logic formalisms, the overall performances are very good. That suggests this approach is pretty promising, and we believe that using direct transformations rules from LTL_f to DFA would give us even better performances.

7.4 Discussion and Future Works

This chapter presented one of the main practical contributions of this thesis. We first described the `Mona` DFA Library, a fundamental building block of our implementation as it provides efficient and optimized implementations of DFAs in semi-symbolic representation and operations over them. Then, we described at high-level `Lydia` and `LydiaSynt`, their purposes, how they are structured and a glance on how they work internally. We presented extensive benchmark from the LTL_f synthesis literature, showing that the approach is very promising and competitive with state-of-the-art tools for LTL_f synthesis and automata construction.

We now discuss future work directions regarding the implementation and better benchmarking of the performances.

7.4.1 Get rid of the `Mona` DFA Library

The `Mona` tool, and in particular the DFA library, has been designed and optimized with in mind the translation procedure of WS1S, and it has shown to be quite efficient and time-proven. For a good survey of implementation secrets in the MONA library that made it successful, you can refer to (Klarlund and Møller, 2001). Nevertheless, a careful rethinking of the data structures involved, and in particular in how the `Mona` DFA is represented, in order to make it suited for our use case of $\text{LTL}_f/\text{LDL}_f$ translation, can bring several benefits to future versions of the `Lydia` tool. Crucially, one limitation is that there is no dynamic reordering of the BDD variables available in the `Mona` BDD library, although it is well known that having such feature very often improves the performances of BDD-based systems (Rudell, 1993). A state-of-the-art library like CUDD (Somenzi, 2015) supports this. Similarly, we can use Algebriac Binary Decision Diagrams (ABDD) (Bahar et al., 1997) as a replacement of the shared multi-terminal BDD of `Mona` in order to represent the transition function.

A completely different approach would be to try to use semi-symbolic automata based on SAT or SMT solvers in order to compactly reason over the symbolic alphabet, instead of using knowledge compilation techniques. The theory of such *symbolic automata*[4], as called by the authors, has been developed in several publications

[4]Note that their automata formalization is still explicit in the state space, and considering the proposed taxonomy in Section 2.5, they would fall in the category of semi-symbolic DFAs but with

(D'Antoni and Veanes, 2017; D'Antoni and Veanes, 2021; Tamm and Veanes, 2018)[5]. Would be very interesting, as a future work, to rely on such theory and see how it compares with the current BDD-based approach.

7.4.2 Improve Experimental Coverage

The benchmark described in Section 7.3 evaluates DFA construction and synthesis from LTL_f formulas taken from the literature of LTL_f synthesis. However, the compositional approach deals with any $\text{LTL}_f/\text{LDL}_f$ formula, potentially coming from any domain of applications. Would be interesting to consider different domains of applications. For example, in the context of software verification (Bouajjani, Habermehl, and Vojnar, 2004), text processing (Alur, D'Antoni, and Raghothaman, 2015), computational linguistics (Mohri, 1997), regular expressions matching (Veanes, Bjørner, and Moura, 2010) and program analysis (Veanes, De Halleux, and Tillmann, 2010).

As we said, our approach bypasses the formalisms WS1S and directly uses the `Mona` DFA library. Would be interesting to encode LDL_f into an equivalent formula of Monadic Second-order Logic (MSO) (which is possible since LDL_f and MSO have the same expressive power), translate the formula into a `Mona` program and give it as input of the `Mona` tool to get the DFA. We can conjecture that this approach is still slower than ours due to the detour to the MSO formalism.

7.4.3 Optimizations

There are several possible optimizations that can be included in our implementation. First of all, we can implement more advanced simplification rules for the input formulas, as done by several other analogous tools such as `Mona` and SPOT (Duret-Lutz et al., 2016). Moreover, we can exploit an hash-consing data structure so to avoid to compute multiple times the same sub-automaton originated from the same subformula. Finally, the last step can be further improved by taking into account *signature equivalences* between formulas. This is an optimization already used by `Mona` for WS1S formulas, in the context of the *DAG construction* of the formula (see Section 4.1 of (Klarlund and Møller, 2001)). That is, at the beginning of the process, the formula is represented using a directed acyclic graph where each node represents either a leaf formula or an operator whose leaves are its operands. If two nodes of the DAG, representing formulas φ_1 and φ_2 respectively, represent the same formula modulo renaming of the variables involved, i.e. they are isomorphic since only the node indices differ, then these nodes can be merged into one. The automaton associated to this node can then be used to compute the DFA of one of the subformulas φ_1 and φ_2, and then compute the other by just reordering the variables.

the transition function using SMT.

[5] https://pages.cs.wisc.edu/~loris/symbolicautomata.html

Part III

Reinforcement Learning with $\mathrm{LTL}_f/\mathrm{LDL}_f$ Specifications

Chapter 8

Background on Reinforcement Learning

In this chapter, we give the background knowledge on the topics of classical Reinforcement Learning and Reinforcement Learning with non-Markovian rewards specified by temporal logic specifications.

The chapter is structured as follows:

- In Section 8.1, we summarize the framework of Reinforcement Learning.
- In Section 8.2, we briefly introduce *Markov Decision Processes* (MDP), the most popular assumed model of the environment in classical Reinforcement Learning.
- In Section 8.3, we revise the main *temporal difference* (TD) learning algorithms, in particular Q-Learning, SARSA and their extension with eligibility traces.
- In Section 8.5, we introduce an important extension of the MDP model, the *Non-Markovian Reward Decision Process* (NMRDP), an important model for later topics in this part of the thesis.
- In Section 8.6, we consider NMRDPs when the non-Markovian reward function is specified by temporal logic specifications, e.g. using $\text{LTL}_f/\text{LDL}_f$ formulas (Brafman, De Giacomo, and Patrizi, 2018).
- Section 8.7 concludes the chapter.

8.1 Reinforcement Learning

Reinforcement Learning (Sutton and Barto, 1998) is a sort of optimization problem where an *agent* interacts with an *environment* and obtains a *reward* for each action he chooses and the new observed state. The task is to maximize a numerical reward signal obtained after each action during the interaction with the environment. The agent does not know a priori how the environment works (i.e. the effects of his actions), but he can make observations in order to know the new state and the reward. Hence, learning is made in a *trial-and-error* fashion. Moreover, it is worth to notice that in many situation reward might not be affected only from the last action but from an indefinite number of the previous actions. In other words, the reward can be *delayed*, i.e. the agent should be able to foresee the effect of his

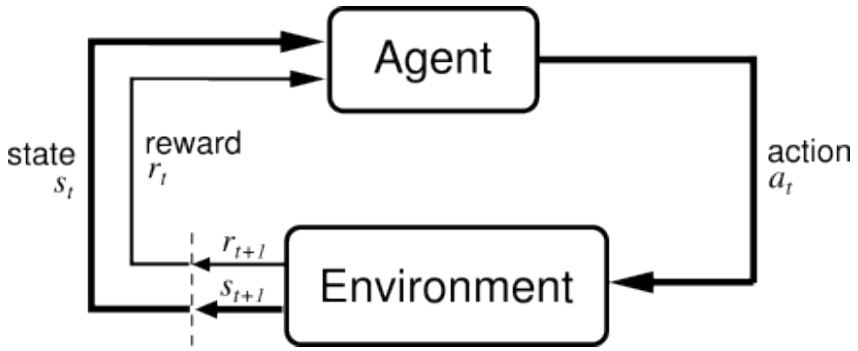


Figure 8.1. The agent and its interaction with the environment in Reinforcement Learning

actions in terms of future expected reward. Figure 8.1 represent the interaction between the agent and the environment in this setting.

In the next sections, we introduce some of the classical mathematical frameworks for RL: Markov Decision Process (MDP) and Non-Markovian Reward Decision Process (NMRDP).

8.2 Markov Decision Process (MDP)

A Markov Decision Process (MDP) $\mathcal{M}$ is a tuple $\langle S, A, T, R, \gamma \rangle$ containing a set of *states* S, a set of *actions* A, a *transition function* $T : S \times A \to Prob(S)$ that returns for every pair state-action a probability distribution over the states, a *reward function* $R : S \times A \times S \to \mathbb{R}$ that returns the reward received by the agent when he performs action a in s and transitions in s', and a *discount factor* γ, with $0 \leq \gamma \leq 1$, that indicates the present value of future rewards. With $T(s, a, s')$ we denote the probability to end in state s' given the action a from s.

The discount factor γ deserves some attention. Its value highly influences the MDP, its solution, and how the agent interprets rewards. Indeed, if $\gamma = 0$, we are in the pure *greedy* setting, i.e. the agent is shortsighted and looks only at the reward that it might obtain in the next step, by doing a single action. The higher γ, the longer the sight horizon, or the foresight, of the agent: the far rewards are taken into account for the current action choice. If $\gamma < 1$ we are in the *finite horizon* setting: namely, the agent is intrinsically motivated to obtain rewards as fast as possible, depending on how γ is far from 1. When $\gamma = 1$ we are in the *infinite horizon* setting, which means that the agent considers far rewards as they can be obtained in the next step. In other words, we may think the agent as *immortal*, since the time the agent spend to reach rewards does not matter anymore.

A *policy* $\rho : S \to A$ for an MDP $\mathcal{M}$ is a mapping from states to actions, and represents a solution for $\mathcal{M}$. Given a sequence of rewards $R_{t+1}, R_{t+2}, \ldots, R_T$, the *expected discounted return* G_t at time step t is defined as:

$$G_t := \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{8.1}$$

where can be $T = \infty$ and $\gamma = 1$ (but not both).

The *value function* of a state s, the *state-value function* $v_\rho(s)$ is defined as the

expected return when starting in s and following policy ρ, i.e.:

$$v_\rho(s) \coloneqq \mathbb{E}_\rho[G_t|S_t = s], \forall s \in S \tag{8.2}$$

Similarly, we define q_ρ, the *action-value function for policy* ρ, as:

$$q_\rho(s,a) \coloneqq \mathbb{E}_\rho[G_t|S_t = s, A_t = a], , \forall s \in S, \forall a \in A \tag{8.3}$$

Notice that we can rewrite 8.2 and 8.3 recursively, yielding the *Bellman equation*:

$$v_\rho(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma v(s')] \tag{8.4}$$

where we used the definition of the transition function:

$$T(s,a,s') = P(s'|s,a) \tag{8.5}$$

We define the *optimal state-value function* and the *optimal action-value function* as follows:

$$v^*(s) \coloneqq \max_\rho v_\rho(s), \forall s \in S \tag{8.6}$$

$$q^*(s,a) \coloneqq \max_\rho q_\rho(s,a), \forall s \in S, \forall a \in A \tag{8.7}$$

Notice that with 8.6 and 8.7 we can show the correlation between $v^*_\rho(s)$ and $q^*_\rho(s,a)$:

$$q^*(s,a) = \mathbb{E}_\rho[R_{t+1} + \gamma v^*_\rho(S_{t+1})|S_t = s, A_t = a] \tag{8.8}$$

We can define a partial order over policies using value functions, i.e. $\forall s \in S.\rho \geq \rho' \iff v_\rho(s) \geq v_{\rho'}(s)$. Now we give the definition of optimal policy:

Definition 8.1. *An* optimal policy ρ^* *is a policy such that* $\rho^* \geq \rho$ *for all* ρ.

Given an MDP $\mathcal{M}$, a typical reinforcement learning problem is the following: find an optimal policy for $\mathcal{M}$, without knowing T and R. Notice that instead of explicit specification of the transition probabilities and rewards, the transition probabilities are accessed through a simulator that is restarted many times from a fixed or uniformly random initial state $s_0 \in S$. We call this way of structuring the learning process *episodic reinforcement learning*. Usually, in episodic reinforcement learning, we require the presence of one or more *goal states* where the simulation of the MDP ends and the task is considered completed, or a maximum time limit T for the number of actions that can be taken by the agent in one single episode, and the overcoming of T determines the end of the episode. Optionally, *failure states* can be also defined, where the episode ends similarly to goal states, but the task is considered failed.

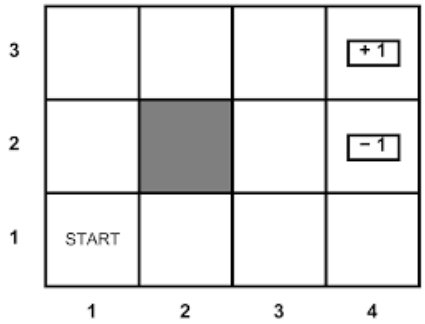
Examples

Many dynamic systems can be modeled as Markov Decision Processes.

Example 8.2 (**Gridworld**). *Perhaps the most simple MDP used as a toy example is* Gridworld*, depicted in Figure 8.2. There are* 3×4 *cells, i.e. states of the MDP* $S = \{s_{11}, s_{12}, \ldots, s_{34}\} \setminus \{s_{22}\}$. *The agent can do four actions:* $A = \{Right, Left, Up, Down\}$. *The initial state is fixed and is* $s_0 = s_{11}$ *and the agent can move in any of the adjacent and free cells from the current state. Assuming*

an episodic task, the goal is to reach s_{34}, and s_{24} represent a failure state. The state transition function T can be deterministic*, i.e. the agent always succeeds in performing actions, or non-deterministic, i.e. the effect of an action is determined by the probabilistic distribution returned by $T(s, a)$. An example of non-deterministic T is to give 90% of success (the agent moves in the chosen direction) and 10% of fail (the agent moves at either the right or left angle to the intended direction). If the move would make the agent walk into a wall (borders of the grid and s_{22}), the agent stays in the same place as before. The reward function $R(s, a, s')$ is defined as -1 if $s' = s_{24}$, as 1 if $s' = s_{34}$, and -0.01 otherwise. The small negative reward given at each transition is a popular mean for reward function design: it is called* step reward *and its purpose is to encourage the agent to finish the episode as fast as possible, with a priority proportional to the absolute value of the reward. The discount factor γ should be strictly higher than 0 because more than one step is needed to reach the goal state.*

Figure 8.2. The Gridworld environment



An example of an optimal policy is shown in Figure 8.3. As the reader can notice, the arrows represent the action that should be taken in a certain cell, in order to maximize the expected return. We observe that the optimal action in s_{13}, according to the policy, is not the one to take the shortest path to the goal, i.e. the Up action . This is because there is a small probability to ens in s_{24}, the failure state, and be punished with a high negative reward. In terms of expected reward, it is better to take the longer path, at the price of collect small negative rewards, but avoiding the risk to fail miserably.

Example 8.3 (**Breakout**)**.** *Breakout is a well-known arcade video game developed*

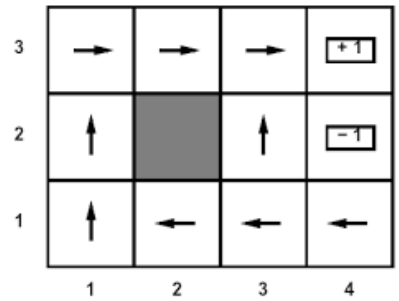


Figure 8.3. An example of optimal policy for the Gridworld environment

.

by Atari. In this work, we implemented a clone of the original Breakout. Figure 8.4 shows a screenshot of the game. On the screen, there is a paddle at the bottom, many bricks at the top arranged in a grid layout with n rows and m columns (in the figure $3 \times 3 = 9$ *bricks), and a ball that is free to move across the screen. The ball bounces when it hits a wall, a brick or the paddle. When the ball hits a brick, that brick is broke down and is removed from the screen. The paddle (the agent) can move left, move right or do nothing. The* goal *is to remove all the bricks while avoiding that the paddle misses the rebound of the ball (*failure*).*

The relevant features are: position of the paddle p_x*, position of the ball* b_x, b_y*, speed of the ball* v_x, v_y *and status of each brick (booleans)* b_{ij}*. This features of the system gives all the needed information to predict the next state from the current state. Hence we can build an MDP where:* S *is the set of all the possible values of the sequence of features* $\langle p_x, b_x, b_y, v_x, v_y, b_{11}, \ldots, b_{nm} \rangle$*,* $A = \{Right, Left, No\text{-}op\}$*, transition function* T *determined by the rules of the game. We give reward* $R(s, a, s') = 10$ *if a particular brick in* s' *has been removed for the first time, plus* 100 *if that brick was the last (i.e. goal reached).*

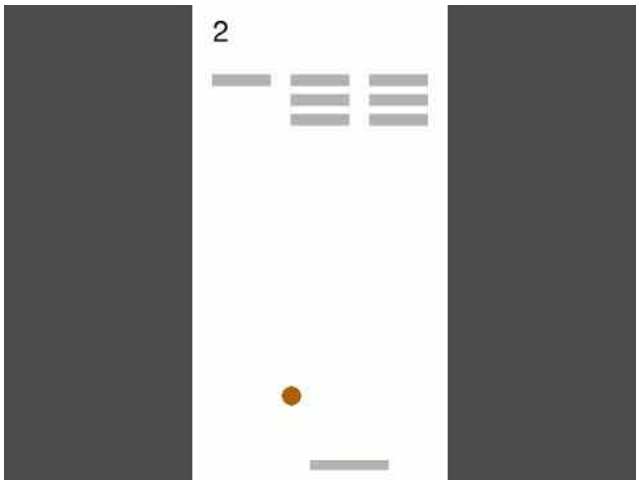


Figure 8.4. A screenshot from the video game BREAKOUT

.

Violation of the Markov property considering a smaller set of features: *Notice that considering a strict subset of the set of features for* S *leads to violating the Markov property of* T*. Indeed, consider the case when we remove* v_x *and* v_y *from the set of features. In this setting, we removed the pieces of information about the dynamics of the system. More precisely, we cannot predict, knowing only the current state, the value of the features* b_x *and* b_y *for the next step, because we do not know where the ball is going (up-left, down-right and so on). In order to correctly predict the next position of the ball, we should know whether earlier in the episode the ball was coming from the bottom or from the top. But this fact clearly shows that the Markovian assumption is violated. Similar arguments apply in the case where we remove the status of the bricks* $b_{11}, \ldots, b_{nm}$*: indeed, if the ball in the next step is near to a brick, knowing about the status of the brick is determinant to predict if the ball will continue its trajectory (the case when the brick is absent) or it will break down the brick and bounce, changing the direction of its motion (the case when the brick is present).*

8.3 Temporal Difference Learning

Temporal difference learning (TD) (Sutton, 1988) refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods sample from the environment, like Monte Carlo (MC) methods, and perform updates based on current estimates, like dynamic programming methods (DP) (Bellman, 1957). We do not discuss MC and DP methods here.

Q-Learning (Watkins, 1989; Watkins and Dayan, 1992) and SARSA (Rummery and Niranjan, 1994; Sutton, 1995) are such a methods. They update $Q(s, a)$, i.e. the estimation of $q^*(s, a)$ at each transition $(s, a) \to (s', r)$. The update rule is the following:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta \tag{8.9}$$

where δ is the *temporal difference*. In Sarsa, it is defined as:

$$\delta = r + \gamma Q(s', a') - Q(s, a) \tag{8.10}$$

whereas in Q-Learning:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \tag{8.11}$$

TD(λ) is an algorithm which uses *eligibility traces*. It is generally believed to outperform simple one-step TD algorithms, since it uses single experiences to update evaluations of multiple state/action pairs that have occurred in the past. The parameter λ refers to the use of an eligibility trace. The algorithm generalizes MC methods and TD learning, obtained respectively by setting $\lambda = 1$ and $\lambda = 0$. Intermediate values of λ yield methods that are often better of the extreme methods. Q-Learning and Sarsa that has been shown before can be rephrased with this new formalism as Q-Learning(0) and Sarsa(0), special cases of Watkin's Q(λ) and Sarsa(λ) respectively. In this setting, Equation 8.9 is modified as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a) \tag{8.12}$$

Where $e(s, a) \in [0, 1]$, the *eligibility of the pair* (s, a), determines how much the temporal difference δ should be weighted. Sarsa(λ) is reported in Algorithm 3, whereas Watkin's Q(λ) in Algorithm 4, both in the variants using *replacing eligibility traces* (see line 9 and line 10, respectively).

8.4 Reward Shaping (RS)

Reward Shaping (Ng, Harada, and Russell, 1999) is a technique to cope with learning on MDPs with sparse rewards, i.e., which occur rarely. The purpose of RS is to guide the agent by exploiting some prior knowledge in the form of additional rewards: $R^s(s, a, s') := R(s, a, s') + F(s, a, s')$, with F the *shaping* function. A desirable requirement of RS is that the additional rewards should not modify the set of optimal policies. This is guaranteed by Potential-Based RS (Ng, Harada, and Russell, 1999) (simply called "Reward Shaping" from now on) which adopts potential functions of the form:

$$F(s.a.s') := \gamma\Phi(s') - \Phi(s) \tag{8.13}$$

In the infinite-horizon case, equation 8.13 guarantees that the set of optimal policies for M and $M^s = \langle S, A, T, R^s, \gamma\rangle$ coincide, for any $\Phi : S \to \mathbb{R}$.

Algorithm 3 Sarsa(λ) (Singh and Sutton, 1996)

Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$ for all s,a
repeat{for each episode}
  initialize s
  Choose a from s using policy derived from Q (e.g. e-greedy)
  repeat{for each step of episode}
    Take action a, observe reward r and new state s'
    Choose a' from s' using policy derived from Q
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow 1$ ▷ replacing traces
    for all s,a **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$, $a \leftarrow a'$
  until state s is terminal
until

Algorithm 4 Watkin's Q(λ) (Watkins, 1989)

Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$ for all s,a
repeat{for each episode}
  initialize s
  Choose a from s using policy derived from Q (e.g. e-greedy)
  repeat{for each step of episode}
    Take action a, observe reward r and new state s'
    Choose a' from s' using policy derived from Q (e.g. e-greedy)
    $a^* \leftarrow \arg\max_a Q(s',a)$ (if a' ties for max, then $a^* \leftarrow a'$)
    $\delta \leftarrow r + \gamma Q(s',a^*) - Q(s,a)$
    $e(s,a) \leftarrow 1$ ▷ replacing traces
    for all s,a **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      if $a' = a^*$ **then**
        $e(s,a) \leftarrow \gamma\lambda e(s,a)$
      else
        $e(s,a) \leftarrow 0$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$, $a \leftarrow a'$
  until state s is terminal
until

Theorem 8.4 ((Ng, Harada, and Russell, 1999)). *Let M be an MDP and let M^s be the same as M but with the reward function R^s, for some Φ. Then, ρ is an optimal policy for M iff ρ is an optimal policy for M^s.*

8.5 Non-Markovian Reward Decision Process (NMRDP)

For some goals, it might be the case that the Markovian assumption of the reward function R – that reward depends only on the current state, and not on history – does not hold. Indeed, for many problems, it is not effective that the reward is limited to depend only on a single transition (s, a, s'); instead, it might be extended to depend on *trajectories* (i.e. $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$), e.g. when we want to reward the agent for some (temporally extended) behaviors, opposed to simply reaching certain states.

This idea of rewarding behaviours has been proposed by (Bacchus, Boutilier, and Grove, 1996) where they defined a new mathematical model, namely Non-Markovian Reward Decision Process (NMRDP), and showed how to construct optimal policies in this case.

In the next subsections, we give the main definitions to reason in this new setting. Then we show the solution proposed in (Bacchus, Boutilier, and Grove, 1996).

8.5.1 Preliminaries

Now follows the definition of NMRDP, which is similar to the MDP definition given in Section 8.2.

Definition 8.5. *A Non-Markovian Reward Decision Process (NMRDP) (Bacchus, Boutilier, and Grove, 1996) $\mathcal{N}$ is a tuple $\langle S, A, T, \bar{R}, \gamma \rangle$ where S, A, T and γ are defined as in the MDP, and $\bar{R} : S^* \to \mathbb{R}$ is the* non-Markovian reward function*, where $S^* = \{\langle s_0, s_1, \ldots, s_n \rangle_{n \geq 0, s_i \in S}\}$ is the set of all the possible traces, i.e. projection of trajectories $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$*

Given a trace $\pi = \langle s_0, s_1, \ldots, s_n \rangle$, the *value of* π is:

$$v(\pi) = \sum_{i=1}^{|\pi|} \gamma^{i-1} \bar{R}(\langle s_0, s_1, \ldots, s_n \rangle) \tag{8.14}$$

where $|\pi|$ denotes the number of transitions (i.e. of actions).

The policy $\bar{\rho}$ in this setting is defined over sequences of states, i.e. $\bar{\rho} : S^* \to A$. The *value of* $\bar{\rho}$ given an initial state s_0 is defined as:

$$v^{\bar{\rho}}(s) = \mathbb{E}_{\pi \sim \mathcal{N}, \bar{\rho}, s_0}[v(\pi)] \tag{8.15}$$

i.e. the expected value in state s considering the distribution of traces defined by the transition function of $\mathcal{N}$, the policy $\bar{\rho}$ and the initial state s_0.

We are interested in two problems, that we will study in the next sections:

- Find an optimal (non-Markovian) policy $\bar{\rho}$ for an NMRDP $\mathcal{N}$ (Definition 8.5);
- Define the non-Markovian reward function for the domain of interest.

8.5.2 Find an optimal policy $\bar{\rho}$ for NMRDPs

The key difficulty with non-Markovian rewards is that standard optimization techniques, most based on Bellman's (Bellman, 1957) dynamic programming principle, cannot be used. Indeed, this requires one to resort to optimization over a policy space that maps histories (rather than states) into actions, a process that would incur a great computational expense. (Bacchus, Boutilier, and Grove, 1996) give the definition of a decision problem *equivalent* to an NMRDP in which the rewards are Markovian. This construction is the key element to solve our problem, i.e. find an optimal policy for an NMRDP.

Equivalent MDP

Now we give the definition of *equivalent* MDP of an NMRDP, and state an important result.

Definition 8.6 ((Bacchus, Boutilier, and Grove, 1996)). *An NMRDP* $\mathcal{N} = \langle S, A, T, \bar{R}, \gamma \rangle$ *is* equivalent *to an extended MDP* $\mathcal{M} = \langle S', A, T', R', \gamma \rangle$ *if there exist two functions* $\tau : S' \to S$ *and* $\sigma : S \to S'$ *such that*

1. $\forall s \in S : \tau(\sigma(s)) = s$;
2. $\forall s_1, s_2 \in S$ *and* $s'_1 \in S'$*: if* $T(s_1, a, s_2) > 0$ *and* $\tau(s'_1) = s_1$*, there exists a unique* $s'_2 \in S'$ *such that* $\tau(s'_2) = s_2$ *and* $T'(s'_1, a, s'_2) = T(s_1, a, s_2)$;
3. *For any feasible trace* $\langle s_0, s_1, \ldots, s_n \rangle$ *of* $\mathcal{N}$ *and* $\langle s'_0, s'_1, \ldots, s'_n \rangle$ *of* $\mathcal{M}$ *associated to the trajectories* $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$ *and* $\langle s'_0, a_0, \ldots, s'_{n-1}, a_{n-1}, s'_n \rangle$*, such that* $\tau(s'_i) = s_i$ *and* $\sigma(s_0) = s'_0$*, we have* $\bar{R}(\langle s_0, s_1, \ldots, s_n \rangle) = R'(s_{n-1}, a_{n-1}, s'_n)$.

Given the Definition 8.6, we give the definition of corresponding policy:

Definition 8.7 ((Bacchus, Boutilier, and Grove, 1996)). *Let* $\mathcal{N}$ *be an NMRDP and let* $\mathcal{M}$ *be the equivalent MDP as defined in Definition 8.6. Let* ρ *be a policy for* $\mathcal{M}$*. The* corresponding policy *for* $\mathcal{N}$ *is defined as* $\bar{\rho}(\langle s_0, \ldots, s_n \rangle) = \rho(s'_n)$*, where for the sequence* $\langle s'_0, \ldots, s'_n \rangle$ *we have* $\tau(s'_i) = s_i \ \forall i$ *and* $\sigma(s_0) = s'_0$

From definitions 8.6 and 8.7, and since that for all policy ρ of $\mathcal{M}$ the corresponding policy $\bar{\rho}$ of $\mathcal{N}$ is such that $\forall s. v_\rho(s) = v_{\bar{\rho}}(\sigma(s))$, the following theorem holds:

Theorem 8.8 ((Bacchus, Boutilier, and Grove, 1996)). *Let* ρ *be an optimal policy for MDP* $\mathcal{M}$*. Then the corresponding policy is optimal for NMRDP* $\mathcal{N}$*.*

The Theorem 8.8 allows us to learn an optimal policy $\bar{\rho}$ for NMRDP by learning a policy ρ over an equivalent MDP, which can be done by resorting on any off-the-shelf algorithm (e.g. see Section 8.3). Moreover, obtaining the corresponding policy for the original NMRDP is straightforward, although in practice is not needed, since it is enough to run the policy ρ over the MDP.

In other words, the problem of finding an optimal policy for an NMRDP reduces to find an optimal policy for an equivalent MDP such that Condition 1, 2 and 3 of Definition 8.6 hold.

8.5.3 Define the non-Markovian reward function $\bar{R}$

To reward agents for (temporally extended) behaviours, as opposed to simply reaching certain states, we need a way to specify rewards for specific trajectories through the state space. Specifying a non-Markovian reward function explicitly is quite hard and unintuitive, impossible if we are in an infinite-horizon setting. Instead, we can define *properties* over trajectories and reward only the ones which satisfy some of them, in contrast to enumerate all the possible trajectories.

Temporal logics presented in Chapter 3 gives an effective way to do this. Indeed, in order to speak about a desired behavior, i.e. fulfillment of properties that might change over time, we can define a *formula* φ (or more formulas) in some suited temporal logic formalism semantically defined over trajectories π, speaking about a set of properties $\mathcal{P}$ such that each state $s \in S$ is associated to a set of propositions ($S \subseteq 2^{\mathcal{P}}$). In this way, a trajectory $\pi = \langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$ is rewarded with r_i iff $\pi \models \varphi_i$, where r_i is the reward value associated to the fulfillment of behaviours signified by φ_i.

8.5.4 Using PLTL

In (Bacchus, Boutilier, and Grove, 1996) the temporal logic formalism is *Past Linear Temporal Logic* (PLTL), which is a past version of LTL (Section 3.1). As explained before, using the declarativeness of PLTL, is possible to specify the desired behaviour (expressed in terms of the properties $\mathcal{P}$) that should be satisfied by the experienced trajectories and reward only them, hence obtaining a non-Markovian reward function. More formally, given a finite set Φ of PLTL *reward formulas*, and for each $\phi_i \in \Phi$ a real-valued reward r_i, the *temporally extended reward function* $\bar{R}$ is defined as:

$$\bar{R}(\langle s_0, s_1, \ldots, s_n \rangle) = \sum_{\phi_i \in \Phi : \langle s_0, s_1, \ldots, s_n \rangle \models \phi_i} r_i \tag{8.16}$$

In order to run the actual learning task, (Bacchus, Boutilier, and Grove, 1996) proposed a transformation from the NMRDP to an equivalent MDP with the state space *expaneded* which allows to label each state $s \in S$. The idea is that the labels should keep track in some way the (partial) satisfaction of the temporal formulas $\phi_i \in \Phi$. A state s in the transformed state space is replicated multiple times, marking the difference between different (relevant) histories terminating in state s.

In this way, they obtained a compact representation of the required history-dependent policy by considering only relevant history, and can produce this policy using computationally-effective MDP algorithms. In other words, the states of the NMRDP can be mapped into those of the expanded MDP, in such a way that corresponding states yield same transition probabilities and corresponding traces have same rewards.

8.6 RL for NMRDP with LTL$_f$/LDL$_f$ Rewards

In this section, we explain the main contribution of this chapter and one of the main contribution of the thesis. We devise a natural extension of the construction explained in (Brafman, De Giacomo, and Patrizi, 2018) for a reinforcement learning task. That is, we show that it is possible to do reinforcement learning for non-Markovian rewards, expressed in LTL$_f$/LDL$_f$ formulas, by applying an extension of the state space of the agent S, analogously to the one described in Section 8.5.4. In the first section, we describe the approach proposed by (Brafman, De Giacomo,

and Patrizi, 2018); then, we observe that the expanded MDP can be used to do reinforcement learning to optimize LTL$_f$/LDL$_f$ non-Markovian rewards.

8.6.1 NMRDP with LTL$_f$/LDL$_f$ rewards

In this section, we explain how to specify non-Markovian rewards with LTL$_f$/LDL$_f$ formulas (instead of PLTL) and how the associated MDP expansion works (Brafman, De Giacomo, and Patrizi, 2018), analogously to what we saw with PLTL (Section 8.5.4).

The temporally extended reward function $\bar{R}$ is similar to Equation 8.16, but instead of using PLTL formula we use LTL$_f$/LDL$_f$ formulas. Formally, given a set of pairs $\{(\varphi_i, r_i)_{i=1}^m\}$ (where φ_i denotes the LTL$_f$/LDL$_f$ formula for specifying a desired behavior, and r_i denotes the reward associated to the satisfaction of φ_i, and given a (partial) trace $\pi = \langle s_0, s_1, \ldots, s_n \rangle$, we define $\bar{R}$ as:

$$\bar{R}(\pi) = \sum_{1 \leq i \leq m : \pi \models \varphi_i} r_i \tag{8.17}$$

For the sake of clarity, in the following we use $\{(\varphi_i, r_i)_{i=1}^m\}$ to denote $\bar{R}$.

Now we describe the MDP expansion for doing learning in this setting, as proposed in (Brafman, De Giacomo, and Patrizi, 2018). Without loss of generality, we assume that every NMRDP $\mathcal{N}$ is reduced into another NMRDP $\mathcal{N}' = \langle S', A', T', R', \gamma \rangle$:

$$\begin{aligned}
S' &= S \cup \{s_{init}\} \\
A' &= A \cup \{start\} \\
T'(s, a, s') &= \begin{cases} 1 & \text{if } \ s = s_{init}, a = start, s' = s_0 \\ 0 & \text{if } \ s = s_{init} \text{ and } (a \neq start \text{ or } s' \neq s_0) \\ T(s, a, s') & \text{otherwise} \end{cases} \\
R'(\langle s_{init}, s_0, \ldots, s_n \rangle) &= R(\langle s_0, s_1, \ldots, s_n \rangle)
\end{aligned} \tag{8.18}$$

and s_{init} is the new initial state. In other words, we prefix to every feasible trajectory $\mathcal{N}$ the pair $\langle s_{init}, start \rangle$, denoting the beginning of the episode. We do this for two reasons: allow to evaluate formulas in s_0 and make it compliant with the most general definition of the reward, namely $R(s, a, s')$, also when there is no true action that is done (i.e. empty trace).

Definition 8.9 ((Brafman, De Giacomo, and Patrizi, 2018))**.** *Given an NMRDP $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)_{i=1}^m, \gamma\} \rangle$ (i.e. with non-Markovian rewards specified by* LTL$_f$/LDL$_f$ *formulas) it is possible to build an $\mathcal{M} = \langle S', A, T', R', \gamma \rangle$ that is* equivalent *(in the sense of Definition 8.6) to $\mathcal{N}$. Denoting with $\mathcal{A}_{\varphi_i} = \langle 2^{\mathcal{P}}, Q_i, q_{i0}, \delta_i, F_i \rangle$ (notice that $S \subseteq 2^{\mathcal{P}}$ and δ_i is total) the* DFA *associated with φ_i (see Chapter 4), the equivalent MDP $\mathcal{M}$ is built as follows:*

- *$S' = Q_1 \times \cdots \times Q_m \times S$ is the set of states;*
- *$T' : S' \times A \times S' \to [0, 1]$ is defined as follows:*

$$Tr'(q_1, \ldots, q_m, s, a, q_1', \ldots, q_m', s') = \begin{cases} Tr(s, a, s') & \text{if } \forall i : \delta_i(q_i, s') = q_i' \\ 0 & \text{otherwise;} \end{cases}$$

- $R' : S' \times A \times S' \to \mathbb{R}$ *is defined as:*

$$R'(q_1, \ldots, q_m, s, a, q'_1, \ldots, q'_m, s') = \sum_{i:q'_i \in F_i} r_i$$

Theorem 8.10 ((Brafman, De Giacomo, and Patrizi, 2018))**.** *The NMRDP* $\mathcal{N} = \langle S, A, T, \{(\varphi_i, r_i)\}_{i=1}^m, \gamma\rangle$ *is equivalent to the MDP* $\mathcal{M} = \langle S', A, T', R', \gamma\rangle$ *defined in Definition 8.9.*

Proof. Recall that every $s' \in S'$ has the form $(q_1, \ldots, q_m, s)$. Define $\tau(q_1, \ldots, q_m, s) = s$. Define $\sigma(s) = (q_{10}, \ldots, q_{m0}, s)$. We have $\tau(\sigma(s)) = s$, hence Condition 1 is verified. Condition 2 of Definition 8.6 is easily verifiable by inspection. For Condition 3, consider a possible trace $\pi = \langle s_0, s_1, \ldots, s_n\rangle$. We use σ to obtain $s'_0 = \sigma(s_0)$ and given s_i, we define s'_i (for $1 \leq i \leq n$) to be the unique state $(q_{1,i}, \ldots, q_{m,i}, s_i)$ such that $q_{j,i} = \delta(q_{j,i-1}, s_i)$ for all $1 \leq j \leq m$. Moreover, we require that, without loss of generality, every trajectory in the new MDP starts from s_{init} and now have a corresponding possible trace of $\mathcal{M}$, i.e., $\pi = \langle s'_0, s'_1, \ldots, s'_n\rangle$. This is the only feasible trajectory of $\mathcal{M}$ that satisfies Condition 3. The reward at $\pi = \langle s_0, s_1, \ldots, s_n\rangle$ depends only on whether or not each formula φ_i is satisfied by π. However, by construction of the automaton $\mathcal{A}_{\varphi_i}$ and the transition function T, $\pi \models \varphi_i$ iff $s'_n = (q_1, \ldots, q_m, s_n)$ and $q_i \in F_i$ □

Let ρ' be a (Markovian) policy for $\mathcal{M}$. It is easy to define an *corresponding* policy on $\mathcal{N}$, i.e., a policy that guarantees the same rewards, by using τ and σ mappings defined in Theorem 8.10 and the result shown in Theorem 8.7.

Lemma 8.11 ((Brafman, De Giacomo, and Patrizi, 2018))**.** *Given an NMRDP* $\mathcal{M}$ *and an equivalent MDP* $\mathcal{M}'$*, every policy* ρ' *for* $\mathcal{M}'$ *has an equivalent policy* $\bar{\rho}$ *for* $\mathcal{M}$ *and viceversa.*

Obviously, typical learning techniques, such as Q-learning or Sarsa, are applicable on the expanded $\mathcal{M}$ and so we can learn an optimal policy ρ for $\mathcal{M}$. Thus, an optimal policy for $\mathcal{N}$ can be learnt on $\mathcal{M}$. Of course, none of these structures is (completely) known to the learning agent, and the above transformation is never done explicitly. Rather, the agent carries out the learning process by assuming that the underlying model is $\mathcal{M}$ instead of $\mathcal{N}$ (applying the fix introduced in Definition 8.18).

Observe that the state space of $\mathcal{M}'$ is the product of the state spaces of $\mathcal{N}$ and $\mathcal{A}_{\varphi_i}$, and that the reward R' is Markovian. In other words, the (stateful) structure of the LTL$_f$/LDL$_f$ formulas φ_i used in the (non-Markovian) reward of $\mathcal{N}$ is *compiled* into the states of $\mathcal{M}$.

Why should we use LDL$_f$

LDL$_f$ formalism (introduced in Section 3.4) has the advantage of *enhanced expressive power* over other proposals, as discussed in (Brafman, De Giacomo, and Patrizi, 2018). Indeed, we move from linear-time temporal logics to LDL$_f$, paying no additional (worst-case) complexity costs. LDL$_f$ can encode in polynomial time LTL$_f$, regular expressions (RE) and the past LTL (PLTL) of (Bacchus, Boutilier, and Grove, 1996). Moreover, LDL$_f$ can naturally represent "procedural constraints" (Baier, Fritz, Bienvenu, et al., 2008), i.e., sequencing constraints expressed as programs, using "if" and "while", hence allowing to express more complex properties.

8.7 Summary

In this chapter, we introduced the topic of Reinforcement Learning, as well as foundational definitions (MDP, policy ρ, state-value function v_ρ) and algorithms (Q(λ), Sarsa(λ)). Then we presented the notion of NMRDP and a technique to build an equivalent MDP, by specifying non-Markovian reward function with $\text{LTL}_f/\text{LDL}_f$ formalisms.

Chapter 9

Restraining Bolts

In this chapter. we investigate on the concept of "*restraining bolt*", as envisioned in Science Fiction. Specifically, we introduce a novel problem in AI. We have two distinct sets of features extracted from the world, one by the agent and one by the authority imposing restraining specifications (the "restraining bolt"). The two sets are apparently unrelated since of interest to independent parties, however they both account for (aspects of) the same world. We consider the case in which the agent is a reinforcement learning agent on the first set of features, while the restraining bolt is specified logically using linear time logic on finite traces $\text{LTL}_f/\text{LDL}_f$ over the second set of features. We show formally, and illustrate with examples, that, under general circumstances, the agent can learn while shaping its goals to suitably conform (as much as possible) to the restraining bolt specifications.

- In Section 9.1, we introduce the context of the work.
- In Section 9.2, we give a formalization of the problem, and prove the main result.
- In Section 9.3, we propose an abstract automata-based reward shaping scheme as a mean to improve learning efficiency.
- In Section 9.4, we present several examples showing applications of restraining bolts.
- Section 9.5 concludes the chapter and discusses future research directions.

The contents of this chapter have been published in the conference paper (De Giacomo, Iocchi, et al., 2019).

9.1 Introduction

This work starts a scientific investigation on the concept of "*restraining bolt*", as envisioned in Science Fiction. A restraining bolt is a "device that restricts a droid's [agent's] actions when connected to its systems. Droid owners install restraining bolts to limit actions to a set of desired behaviors."[1] The concept of restraining bolt introduces a new problem in AI. We have two distinct representations of the world, one by the agent and one by the authority imposing restraining specifications, i.e., the bolt. Such representations are apparently unrelated as developed by independent

[1] https://www.starwars.com/databank/restraining-bolt

parties, but both model (aspects of) the same world. We want the agent to conform (as much as possible) to the restraining specifications, even if these are not expressed in terms of the agent's world representation.

Studying this problem from a classical Knowledge Representation perspective (Reiter, 2001) would require to establish some sort of "glue" between the representation by the agent and that by the restraining bolt. Instead, we bypass dealing with such a "glue" by studying this problem in the context of Reinforcement Learning (RL) (Puterman, 1994; Sutton and Barto, 1998), which is currently of great interest to develop components with forms of decision making, possibly coupled with deep learning techniques (Mnih et al., 2015; Silver et al., 2017).

Specifically, we consider an agent and a restraining bolt of different nature. The *agent* is a reinforcement learning agent whose "model" of the world is a hidden, factorized, Markov Decision Processes (MDP) over a certain set of world features. That is, the state is factorized in a set of features observable to the agent, while transition function and reward function are hidden. The *restraining bolt* consists in a logical specification of traces that are considered desirable. The world features that are used to represent states in these traces are disjoint from those used by the agent. More concretely such specifications are expressed in full-fledged temporal logics over finite traces, LTL_f and its extension LDL_f (De Giacomo and Vardi, 2013; De Giacomo and Rubin, 2018; Brafman, De Giacomo, and Patrizi, 2018). Notice that the restraining bolt does not have an explicit model of the dynamics of the world, nor of the agent. Still it can assess if a given trace generated by the execution of the agent in the world is desiderabile, and give additional rewards when it does.

The connection between the agent and the restraining bolt is loose: the bolt provides additional reward to the agent and only needs to know the order of magnitude of the original rewards of the agent to suitably fix a scaling factor[2] for its own additional rewards. In addition, it provides to the agent additional information to allow the agent to know at what stage of the satisfaction of temporal formulas the world is so that the agent can choose its policy accordingly. Without them, the agent would not be able to act differently at different stages to get the rewards according to the temporal specifications.

The main result of this chapter is that, in spite of the loose connection between the two models, under general circumstances, the *agent can learn to act so as to conform as much as possible to the* $\text{LTL}_f/\text{LDL}_f$ *specifications*. Observe that we deal with two separate representations (i.e., two distinct sets of features), one for the agent and one for the bolt, which are apparently unrelated, but in reality, correlated by the world itself, cf., (Brooks, 1991). The crucial point is that, in order to perform RL effectively in presence of a restraining bolt *such a correlation does not need to be formalized*.

For example, consider a service robot serving drinks and snacks at a party. The robot knows the locations where it can grasp drink and snack items and the locations of people that can receive such items. The robot can execute actions to move in the environment, to grasp objects and to deliver them to people. With rewards associated to effective deliver, the robot can learn how to serve something to a specific person. Now, suppose we want to impose the following restraining bolt specification: *serve exactly one drink and one snack to every person, and do not serve alcoholic drinks to minors*. To express this specification (e.g., as an $\text{LTL}_f/\text{LDL}_f$ formula), a representation of the status of each person (i.e., identity, age and received items) is needed, even though these features are *not* available to the learning agent

[2]Note that finding the right scaling factor is an important issue in RL (Simsek and Barto, 2006), but out of the scope of this work.

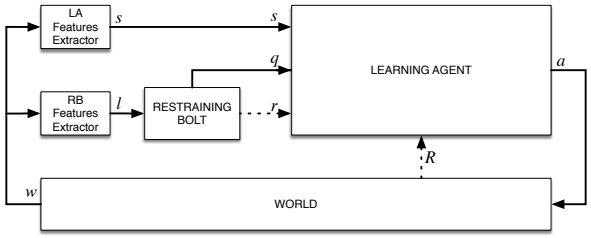


Figure 9.1. Learning Agent and Restraining Bolt

(but only to the restraining bolt).

Notice that the presence of LTL$_f$/LDL$_f$ specification makes the whole system formed by the agent and the restraining bolt non-Markovian. Recently, interest in non-Markovian Reward Decision Processes NMRDPs (Bacchus, Boutilier, and Grove, 1996; Whitehead and Lin, 1995), and in particular on expressing rewards using linear-time temporal logic has been revived and motivated by the difficulty in rewarding complex behaviors directly on MDPs (Littman, 2015a; Littman et al., 2017). In this context, the use of linear time logics over finite traces such as LTL$_f$ or its extension LDL$_f$ has been recently advocated (Camacho, Chen, et al., 2017a; Brafman, De Giacomo, and Patrizi, 2018; Icarte, Klassen, et al., 2018b). Notably, both LTL$_f$ and LDL$_f$ formulas can be transformed into deterministic finite state automata tracking the stage of satisfaction of the formulas (De Giacomo and Vardi, 2013). This, in turn, allows for transforming an NMRDP with non-Markovian LTL$_f$/LDL$_f$ rewards into an equivalent MDP over an extended state space, obtained as the crossproduct of the states of the NMRDP and the states of the automaton. This transformation is of particular interest in RL with temporally specified rewards expressed in LTL$_f$/LDL$_f$, since it allows to do RL on an equivalent MDP whose (optimal) policies are also (optimal) policies for the original problem, and viceversa (Brafman, De Giacomo, and Patrizi, 2018). This provides the basis for our development here.

Summarizing, in this chapter, we set the framework for the problem of restraining bolt in RL context and provide proofs and practical evidence, through various examples, that an RL agent can learn policies that optimize conformance to the LTL$_f$/LDL$_f$ restraining specifications, without including in the agent's state space representation the features needed to evaluate the LTL$_f$/LDL$_f$ formula. Our work can also be seen as a contribution to the research providing safety guarantees to AI techniques based on learning. We take up this point in a brief discussion at the end of the chapter.

9.2 RL with LTL$_f$/LDL$_f$ restraining specifications

We now focus on the restraining bolt problem, i.e., how to do RL with restraining specifications expressed in LTL$_f$/LDL$_f$.

We are given:

- A **learning agent** modeled by the MDP $M_{ag} = \langle S, A, Tr_{ag}, R_{ag} \rangle$, with S denoting the set of configurations of the features observable by the agent, transitions Tr_{ag} and rewards R_{ag} hidden, but sampled from the environment.
- A **restraining bolt** $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^{m} \rangle$ where:
  - $\mathcal{L} = 2^{\mathcal{F}}$ is the set of possible fluents' configurations observable by the

bolt. Fluents in $\mathcal{F}$ are not among the features that form the states S of the learning agent M_{ag}.

- $\{(\varphi_i, r_i)\}_{i=1}^m$ is a set of *restraining specifications* with
  - φ_i, an LTL$_f$/LDL$_f$ formula over $\mathcal{F}$. Each φ_i selects sequences of fluents' configurations $\ell_1, \cdots, \ell_n$ ($\ell_k \in \mathcal{L}$) whose relationship with the sequences of states $s_1, \ldots, s_n$ ($s_k \in S$) of M_{ag} is unknown.
  - r_i, the reward associated with φ_i. A reward r_i is assigned to sequences of configurations $\ell_1, \cdots, \ell_n$ satisfying φ_i.

The agent receives rewards based on R_{ag} and the pairs (φ_i, r_i). In fact, often we have to handle tasks of episodic nature. That is, the world can reach a configuration in which no action can change its configuration nor generate new rewards, e.g., a final configuration in a game. In this case we assume that the restraining bolt fluents $\mathcal{F}$ include a special fluent *Done* that denotes reaching the final configuration. This fluent can be used in LTL$_f$/LDL$_f$ formulas to reward the agent only at the end of the episode. When the episode ends and a new episode is started, a new trace is generated on which LTL$_f$/LDL$_f$ formulas are evaluated again.

Notice that while the agent can see the features that the reward R_{ag} depends on, it cannot see those that affect r_i. Both S and $\mathcal{L}$ are features' configurations, in the sense of representing world properties. However, they capture different facets of the world. Let W be the set of *real world states*. A *feature* is a function $f_j : W \to D_j$ that maps world states to another domain D_j, such as reals, enumerations, booleans, etc. The *feature vector* of a world state w_h is the vector $\mathbf{f}(w_h) = \langle f_1(w_h), \ldots, f_d(w_h)\rangle$ of feature values corresponding to w_h. Given a world state w_h, the corresponding *configuration s_h of the learning agent M_{ag}* consists in those components of $\mathbf{f}(w_h)$ that produce the agent's state, while the corresponding *configuration of fluents ℓ_h* is formed by the components that assign truth values to the fluents. That is, a subset of the world features describes the agent states s_h and another subset (for simplicity, assumed disjoint from the previous one) is used to evaluate the fluents in ℓ_h. Hence, a sequence $w_1, \ldots, w_n$ of world states defines both a sequence of learning agent states $s_1, \ldots, s_n$ and a sequence of fluent configurations $\ell_1, \ldots, \ell_n$. While R_{ag} depends on the former, each φ_i and r_i depend on the latter. Consequently, by executing a policy and hence by repeatedly choosing actions in A, the agent visits a sequence of world states, collecting for each of them, the sum of the rewards R_{ag} and r_i. The point to resolve is defining on the base of which observations the agent can choose its next actions. Obviously, the agent can in principle accumulate all its observations $s_1, \ldots, s_n$, but on the other hand it cannot see the fluents configurations $\ell_1, \ldots, \ell_n$. In order to drive the learning process, we want to equip the agent with some means to establish the stage of satisfaction of the formulas φ_i. Such a notion, as mentioned above, can be captured by considering *the* minimal DFA $\mathcal{A}_{\varphi_i} = \langle 2^{\mathcal{P}}, Q_i, q_{i0}, \delta_i, F_i\rangle$ corresponding to formula φ_i. Notice that such a DFA is unique. Hence we equip the agent with additional observable features $Q_1 \times \ldots \times Q_m$ corresponding to the states of $\mathcal{A}_{\varphi_i}$. Such features are going to be provided by the restraining bolt.[3] Note that this does not give away fluents configurations $\ell_1, \ldots, \ell_n$ which remain hidden to the agent, see Figure9.1. Moreover, to keep the RL agent's state representation independent from the bolt, only an encoding of the features $Q_1 \times \ldots \times Q_m$ are

[3]Notice that coming up with the $Q_1, \ldots, Q_n$ and assigning the rewards to some of them, while can perhaps be possible in very simple cases, without a principled and systematic technique as the one presented here it is virtually impossible. Indeed, to express directly LTL$_f$/LDL$_f$ properties in the MDP, one may need exponential additional features, assuming a factorized representation, since the corresponding DFA may be doubly exponential in the formula.

provided (e.g., an m-tuple of integer numbers). In other words, the RL agent does not need to know semantic information about the state of the bolt (i.e., the state of satisfactions of the LTL$_f$/LDL$_f$ formulas), thus Q_i does not need to be expressed in terms of fluents in $\mathcal{F}$. Indeed, any encoding allowing to distinguish different states of the DFA each other is sufficient.

Hence, in general, we consider possibly non-Markovian policies of the form

$$\bar{\rho} : (Q_1 \times \ldots \times Q_m \times S)^* \to A$$

and thus define the expected (discounted) cumulative reward of a possibly non-Markovian policy $\bar{\rho}$ as the expected reward of infinite traces starting in the initial state s_0, induced by the policy itself (obtained as the expected sum of the collected rewards R_{ag} and r_i).

Problem definition. (An instance of) the *RL problem with* LTL$_f$/LDL$_f$ *restraining specifications* is a pair $M^{rb}_{ag} = \langle M_{ag}, RB \rangle$, where: $M_{ag} = \langle S, A, Tr_{ag}, R_{ag} \rangle$ is a learning agent with Tr_{ag} and R_{ag} hidden, and $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^m \rangle$ is a restraining bolt formed by a set of LTL$_f$/LDL$_f$ formulas φ_i over $\mathcal{L}$ with associated rewards r_i. A solution to the problem is a (possibly non-Markovian) policy $\bar{\rho} : (Q_1 \times \ldots \times Q_m \times S)^* \to A$ that maximizes the expected cumulative reward.

To devise a solution technique, we assume that the agent actions in A induce a transition distribution over the features and fluents configuration, i.e.:[4]

$$Tr^{rb}_{ag} : S \times \mathcal{L} \times A \to Prob(S \times \mathcal{L}).$$

Such a transition distribution, together with the initial values of the fluents ℓ_0 and of the agent state s_0, allow us to describe a probabilistic transition system accounting for the dynamics of the fluents and agent states. Moreover, when Tr^{rb}_{ag} is projected on S only, i.e., the $\mathcal{L}$ components are marginalized, we get Tr_{ag} of M_{ag}. Obviously, both Tr^{rb}_{ag} and Tr_{ag} are hidden to the learning agent. On the other hand, in response to an agent action a_h performed in the current state w_h (in the state s_h of the agent and the configuration ℓ_h of the fluents), the world changes into w_{h+1} from which s_{h+1} and ℓ_{h+1} are obtained. This is all we need to proceed.

Given $M^{rb}_{ag} = \langle M_{ag}, RB \rangle$ with $M_{ag} = \langle S, A, Tr_{ag}, R_{ag} \rangle$ and $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^m \rangle$, we define an NMRDP $M^n_{ag} = \langle S \times \mathcal{L}, A, Tr^{rb}_{ag}, \{(\varphi_i, r_i)\}_{i=1}^m \cup \{(\varphi_s, R_{ag}(s, a, s'))\}_{s \in S, a \in A, s' \in S} \rangle$, where:

- states are pairs (s, ℓ) formed by an agent configuration s and a fluents configuration ℓ;
- φ_i are as before;
- $\varphi_s = \Diamond(s \wedge a \wedge \bigcirc(Last \wedge s'))$;
- Tr^{rb}_{ag}, r_i and $R_{ag}(s, a, s')$ are hidden and sampled from the environment.

Formulas φ_i are as before, in particular they are continuously evaluated on the (partial) trace produced so far. Though, they may use the special fluent *Done* to give the reward associated to the formula at the end of the episode (modulo reward shaping). Formulas $\Diamond(s \wedge a \wedge \bigcirc(Last \wedge s'))$, one per (s, a, s'), which require that both states s and action a are followed by s', are evaluated at the end of

[4] Notice that this assumption is quite loose, as we can arbitrarily enlarge $\mathcal{L}$ to define Tr^{rb}_{ag}. In the construction below only the fluents in $\mathcal{L}$ that occur in the LTL$_f$/LDL$_f$ formulas play an active role.

the current (partial) trace (recall that *Last* denotes the last element of the trace, c.f. Preliminaries). In this case, the reward $R_{ag}(s, a, s')$ from M_{ag} associated with (s, a, s') is given.[5]

Observe that policies for M^n_{ag} have the form $(S \times \mathcal{L})^* \to A$ which needs to be restricted to have the form required by our problem M^{rb}_{ag}. A policy $\bar{\rho} : (S \times \mathcal{L})^* \to A$ *has the form* $\bar{\rho} : (Q_1 \times \ldots \times Q_m \times S)^* \to A$ when $\bar{\rho}(\langle s_1, \ell_1\rangle \cdots \langle s_n, \ell_n\rangle) = \bar{\rho}(\langle q_{11}, \ldots, q_{m1}, s_1, \rangle \cdots \langle q_{1n}, \ldots, q_{mn}, s_n, \rangle)$ with $q_{ij} = \delta_j(\ell_1, \ldots, \ell_i, q_{j0})$. In other words, a policy $\bar{\rho} : (S \times \mathcal{L})^* \to A$ has the form $\bar{\rho} : (Q_1 \times \ldots \times Q_m \times S)^* \to A$ when the fluents $\mathcal{L}$ are not directly accessible but are used only to progress the DFAs $\mathcal{A}_{\varphi_i}$ corresponding to formulas φ_i. We can now state the following result.

Lemma 9.1. *RL with* LTL$_f$/LDL$_f$ *restraining specifications* $M^{rb}_{ag} = \langle M_{ag}, RB\rangle$ *with* $M_{ag} = \langle S, A, Tr_{ag}, R_{ag}\rangle$ *and* $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}^m_{i=1}\rangle$ *can be reduced to RL over the NMRDP* $M^n_{ag} = \langle S \times \mathcal{L}, A, Tr^{rb}_{ag}, \{(\varphi_i, r_i)\}^m_{i=1} \cup \{(\varphi_s, R_{ag}(s, a, s'))\}_{s\in S, a\in A, s'\in S}\rangle$*, by restricting the policy to learn to have the form* $\bar{\rho} : (Q_1 \times \ldots \times Q_m \times S)^* \to A$*.*

Proof. By construction. □

As a second step, we apply the construction of the previous section and obtain a new MDP learning agent. In such construction, because of their triviality, we do not need to keep track of the state of the automata associated with each φ_s, but just offer the reward $R_{ag}(s, a, s')$ associated with (s, a, s'). Instead, we do need to keep track of state of each DFA $\mathcal{A}_{\varphi_i} = \langle 2^{\mathcal{P}}, Q_i, q_{i0}, \delta_i, F_i\rangle$ corresponding to φ_i. Hence, from M^n_{ag}, we obtain an MDP $M'_{ag} = \langle S', A, Tr'_{ag}, R'_{ag}\rangle$ where:

- $S' = Q_1 \times \cdots \times Q_m \times S \times \mathcal{L}$ is the set of states;
- $Tr'_{ag} : S' \times A \times S' \to [0, 1]$ is defined as follows:

$$Tr'_{ag}(q_1, \ldots, q_m, s, \ell, a, q'_1, \ldots, q'_m, s', \ell') = \begin{cases} Tr_{ag}(s, \ell, a, s', \ell') & \text{if } \forall i : \delta_i(q_i, \ell') = q'_i \\ 0 & \text{otherwise;} \end{cases}$$

- $R'_{ag} : S' \times A \times S' \to \mathbb{R}$ is defined as:

$$R'_{ag}(q_1, \ldots, q_m, s, \ell, a, q'_1, \ldots, q'_m, s', \ell') = \sum_{i:q'_i\in F_i} r_i + R_{ag}(s, a, s')$$

Observe that, besides the rewards $R_{ag}(s, a, s')$ of the original learning agent, the environment now offers the rewards r_i associated with the formulas φ_i, thus guiding the agent towards the satisfaction of the φ_i (by progressing correctly the corresponding DFAs $\mathcal{A}_{\varphi_i}$).

By Theorem 8.10, it follows that the NMRDP M^n_{ag} and the MDP M'_{ag} are equivalent. Hence, by Lemma 8.11, any policy of M^n_{ag} has an equivalent policy (hence guaranteeing the same reward) in M'_{ag}, and viceversa. We can thus learn a policy on M'_{ag} instead of M^n_{ag}. We can thus refine Lemma 9.1 into the following.

Lemma 9.2. *RL with* LTL$_f$/LDL$_f$ *restraining specifications* $M^{rb}_{ag} = \langle M_{ag}, RB\rangle$ *with* $M_{ag} = \langle S, A, Tr_{ag}, R_{ag}\rangle$ *and* $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}^m_{i=1}\rangle$ *can be reduced to RL over the*

[5] Notice that we have as many of such formulas as transitions (s, a, s'), this causes an exponential blow-up being S factorized in features. However, we will get rid of them later.

MDP $M'_{ag} = \langle S', A, Tr'_{ag}, R'_{ag}\rangle$, by restricting the policy to learn to have the form $Q_1 \times \ldots \times Q_n \times S \to A$.

Proof. By construction. □

This Lemma allows for restricting, without loss of generality, non-Markovian policies $(Q_1 \times \ldots \times Q_n \times S)^* \to A$ to Markovian policies $Q_1 \times \ldots \times Q_n \times S \to A$.

As a last step, we solve the original RL task on M^{rb}_{ag} by performing RL on a new MDP $M^{q}_{ag} = \langle Q_1 \times \cdots \times Q_m \times S, A, Tr''_{ag}, R''_{ag}\rangle$, where:

- The transition distribution Tr''_{ag} is the marginalization of Tr'_{ag} wrt $\mathcal{L}$, and is unknown;
- The reward R''_{ag} is defined as:

$$R''_{ag}(q_1, \ldots, q_m, s, a, q'_1, \ldots, q'_m, s') = \sum_{i:q'_i \in F_i} r_i + R_{ag}(s, a, s').$$

- The states q_i of the DFAs $\mathcal{A}_{\varphi_i}$ are progressed correctly by the environment.

Thus, we finally obtain our main result.

Theorem 9.3. *RL with LTL$_f$/LDL$_f$ restraining specifications $M^{rb}_{ag} = \langle M_{ag}, RB\rangle$ with $M_{ag} = \langle S, A, Tr_{ag}, R_{ag}\rangle$ and $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^m\rangle$ can be reduced to RL over the MDP $M^{q}_{ag} = \langle Q_1 \times \cdots \times Q_m \times S, A, Tr''_{ag}, R''_{ag}\rangle$ and optimal policies ρ^{new}_{ag} for M^{rb}_{ag} can be learned by learning corresponding optimal policies for M^{q}_{ag}.*

Proof. For brevity we use $\mathbf{q}$ to denote $q_1, \ldots, q_m$. By Lemma 9.2 we can focus on RL over the MDP $M'_{ag} = \langle S', A, Tr'_{ag}, R'_{ag}\rangle$ under the restriction that the policy to learn has the form $Q_1 \times \ldots \times Q_n \times S \to A$.

Notice that from the definitions of R' and R'', we have that for all $\ell, \ell' \in \mathcal{L}$, $R'(\mathbf{q}, s, \ell, a, \mathbf{q}', s', \ell') = R''(\mathbf{q}, s, a, \mathbf{q}', s') = \sum_{i:q'_i \in F_i} r_i + R(s, a, s')$. The crux of the proof is to show that for any optimal policy ρ the values $v^\rho(\mathbf{q}, s, \ell)$ of the state value function for $\mathcal{M}'_{ag}$ do not depend on ℓ. That is, it is necessary that $\forall \ell_1, \ell_2. v^\rho(q_1, \ldots, q_m, s, \ell_1) = v^\rho(q_1, \ldots, q_m, s, \ell_2)$.

To see this, let $T'_{ag}(s, a, s') = P(s'|s, a)$, then the Bellman equation in our case is:

$$v^\rho(\mathbf{q}, s, \ell) = \sum_{\mathbf{q}', s', \ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, \ell, a)[R'(\mathbf{q}, s, \ell, a, \mathbf{q}', s', \ell') + \gamma v^\rho(\mathbf{q}', s', \ell')].$$

By using the equality between R' and R'' we have:

$$v^\rho(\mathbf{q}, s, \ell) = \sum_{\mathbf{q}', s', \ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, \ell, a)[R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v^\rho(\mathbf{q}', s', \ell')].$$

On the other hand, observe that we can compute $\mathbf{q}'$ from $\mathbf{q}$ and ℓ', that is we do not need ℓ. Hence: $P(\mathbf{q}', s', \ell' | \mathbf{q}, s, \ell, a) = P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)$. So we can write:

$$v^\rho(\mathbf{q}, s, \ell) = \sum_{\mathbf{q}', s', \ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)[R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v^\rho(\mathbf{q}', s', \ell')].$$

At this point, we see that v^ρ does not depend from ℓ, hence we can safely drop ℓ as argument for v_ρ. Indeed, we get:

$$v^\rho(\mathbf{q}, s) = \sum_{\mathbf{q}', s'} [R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v^\rho(\mathbf{q}', s')] \sum_{\ell'} P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)$$

and by marginalizing the distribution $P(\mathbf{q}', s', \ell' | \mathbf{q}, s, a)$ over ℓ', we get:

$$v^\rho(\mathbf{q}, s) = \sum_{\mathbf{q}', s'} P(\mathbf{q}', s' | \mathbf{q}, s, a)[R''(\mathbf{q}, s, a, \mathbf{q}', s') + \gamma v^\rho(\mathbf{q}', s')].$$

This is Bellman's equation for M^q_{ag}, hence the thesis. □

This theorem provides us with a technique to learn the optimal policy for RL with $\text{LTL}_f/\text{LDL}_f$ restraining specification by minimizing the intervention on the learning agent: essentially we need to feed it with the rewards r_i at suitable times, and we need to allow the learning agent to keep track of the stage of satisfaction of the restraining bolt formulas by feeding it with new features for $Q_1, \ldots, Q_n$.

9.3 Automata-based reward shaping

Reward shaping is a well-known technique to guide the agent during the learning process and so reduce the time needed to learn. The possibility of using reward shaping in the context of RL for $\text{LTL}_f/\text{LDL}_f$ rewards has been exploited in (Camacho, Chen, et al., 2017a). The idea is to supply additional rewards in a proper manner such that the optimal policy is the same of the original MDP. Formally, the original reward $R(s, a, s')$ is replaced by $R'(s, a, s') = R(s, a, s') + F(s, a, s')$, where $F(s, a, s')$ is the *shaping reward function*. In (Ng, Harada, and Russell, 1999) it has been shown that potential-based reward shaping of the form $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$, for some $\Phi : S \to \mathbb{R}$, is a necessary and sufficient condition for policy invariance under this kind of reward transformation, i.e. the optimal and near-optimal MDP solutions are preserved.

We observe that, the use of reward shaping when using $\text{LTL}_f/\text{LDL}_f$ rewards φ can be automatized. Given a $\text{LTL}_f/\text{LDL}_f$ formula φ, we build the associated DFA $\mathcal{A}_\varphi$. This operation is made *off-line*, i.e. before the learning process. Then we associate automatically to the states of the DFA a potential function $\Phi(q)$ whose value decreases proportionally with the minimum distance between the automaton state q and any accepting state. The potential function gives a positive reward when the agent performs an action leading to a q' that is one step closer to an accepting state, and a negative one in the opposite case. Moreover, with $\gamma < 1$, a penalty is given if $\Phi(q) = \Phi(q')$.

Reward shaping can also be used when the DFAs of the $\text{LTL}_f/\text{LDL}_f$ formulas are constructed *on-the-fly* (Brafman, De Giacomo, and Patrizi, 2018) so as to avoid to compute the entire automaton off-line. To do so we can rely on *dynamic reward shaping* (Devlin and Kudenko, 2012). The idea is to build $\mathcal{A}_\varphi$ progressively while learning. During the learning process, at every step, the value of the fluents $\ell \in \mathcal{L}$ is observed and the successor state q' of the current state q of the DFA on-the-fly is computed. Then, the transition and the new state just observed are added into the "built" automaton at time t, $\mathcal{A}_{\varphi,t}$, yielding $\mathcal{A}_{\varphi,t'}$. The potential function Φ for $\mathcal{A}_{\varphi,t'}$ is recomputed for the new version of the automaton. In this case, the shaping reward function takes the following form:

$$F(q, t, a, q', t') = \gamma\Phi(q', t') - \Phi(q, t)$$

Figure 9.2. Experimental scenarios: BREAKOUT, SAPIENTINO, COCKTAILPARTY

where $\Phi(q,t)$ is the same of the off-line variant (with some additional heuristics) but computed on the automaton $\mathcal{A}_{\varphi,t}$. Optimality and near-optimality guarantees are still preserved as explained in (Devlin and Kudenko, 2012).

Theorem 9.4. *Automata-based reward shaping, both in off-line and on-the-fly variants, preserves optimality and near-optimality of the MDP solutions.*

Proof. For the off-line case, the shaping-reward function Φ is, by construction, potential based, hence fulfilling the premises of theorems in (Ng, Harada, and Russell, 1999) and (Grześ, 2017). Also for the on-the-fly variant, we observe that our construction is compliant with the requirements shown in (Devlin and Kudenko, 2012). □

9.4 Implementation and Examples

Implementation of agents learning policies with restraining specifications is performed by assuming a learning phase in simulation and an execution phase on the real world. The learning phase is obtained by combining three software components: 1) a simulator of the dynamic system, 2) a restraining bolt (RB) process, 3) a reinforcement learning (RL) agent. All these components are modular (i.e., they can be properly connected with each other or replaced by other similar components). The simulator is responsible for computing the evolution of the dynamic system under study: it receives decisions (actions to be executed) by the RL agent and communicates: i) the current state of the system to both the RL agent and the RB process, and ii) the current reward value to the RL agent. The RB process receives the current state from the simulator, evaluates the $\text{LTL}_f/\text{LDL}_f$ formulas denoting the restraining specifications and sends to the RL agent an encoding of the progress of the DFA representing the formulas and reward values associated to their evolution. Finally, the RL agent receives the simulator state, the RB state, and the rewards and decides the actions to be executed, while computing an optimal policy. By using such a simulator, the RL agent can learn a policy that maximizes the cumulative discounted reward by taking into account the rewards from both the environment and the RB. In general, when enough training is allowed, the computed policy, when executed on the real world, will satisfy the RB specifications.

As mentioned, the RL agent and the RB process have different sensors to perceive different aspects of the state of the world (or of the simulator). So we assume that they are implemented with real sensors (when attached to the real world) and corresponding virtual sensors (when attached to the simulator). We also assume that the simulator is able to model all the relevant evolutions of the world that are needed to learn the specific task with restraining specifications.

Next we show the implementation of such components in three examples (Figure 9.2). The first one uses a video-game simulator, while the other two consider robotic tasks and their corresponding models in a simulator. The core software for the RL agent and for the RB process are domain-independent, while the (virtual) sensors and the $\text{LTL}_f/\text{LDL}_f$ specifications are domain-dependent. Since all examples

are of episodic nature, the learning phase is managed by an execution system that resets episodes when any of the following conditions is verified: 1) a state of the DFA where the formula is satisfied is reached, 2) a failure state of the DFA (i.e., a state from which it is not possible to satisfy any formula) is reached, 3) a maximum number of actions have been executed (to avoid infinite loops).

To speed up learning, the implementation of the bolt monitors the progress of the DFA corresponding to the restraining specifications and applies a kind of reward shaping by exploiting the DFA structure

Through reward shaping we can anticipate part of the reward coming from temporal specifications without waiting for the formulas to become true.

Each experiment (i.e., a sequence of episodes to learn a policy) terminates after a time limit that is different for each problem (see next sections) and chosen to guarantee that a policy consistent with the specifications is always found, although in general not optimal. All the problems described below have been solved with n-step Sarsa algorithm, configured with $\gamma = 0.999$, $\epsilon = 0.2$, $n = 100$. The trend of the solutions is anyway not sensitive to these parameters.

Algorithms have been implemented as single-thread non-optimized Python procedures, in a modular and abstract way to operate on every problem. More details about the experimental configurations, source code of the implementation allowing for reproducing the results contained in this paper, and videos of the found policies are available in www.diag.uniroma1.it/restraining-bolt.

Breakout. BREAKOUT has been widely used to demonstrate RL approaches. The goal of the agent is to control the paddle in order to drive a ball to hit all the bricks in the screen. In this example, we have considered two agents with different abilities: MOVE: the agent moves sideways to bounce the ball; MOVE +FIRE: the agent can both move and fire straight up to remove bricks. Agent's state representation uses the following features: f_x: x position of the paddle; $f_{bx}, f_{by}, f_{dx}, f_{dy}$: position and direction of movement of the ball[6]. Reward is given to the agent when a brick is hit. With this specification a RL algorithm can find a policy to remove all the bricks and complete the game for both the agents.

Restraining bolt. We want to provide the agents with the following specification: *the bricks must be removed from left to right*, i.e., all the bricks in column i must be removed before completing any other column $j > i$. This specification can be expressed with an $\text{LTL}_f/\text{LDL}_f$ formula and to evaluate such a formula, the bolt needs a representation $f_{r(i,j)}$ of the status of each brick $r_{i,j}$ (present or removed). The agents, after receiving in input from the bolt an encoding of the status of the $\text{LTL}_f/\text{LDL}_f$ formula and associated rewards, can use the same RL algorithm to learn a new policy that will complete the task (i.e., remove all the bricks) following the restraining bolt specification (i.e., from left to right).

Notice that the same restraining bolt is applied to the two different agents and they will both learn the behavior specified by the $\text{LTL}_f/\text{LDL}_f$ formula, obviously with different policies. Rows 1 and 2 in Figure 9.3 show the results of two experiments in the Breakout scenario with the following configurations: Breakout 4x6 MOVE + FIRE (5 minutes), Breakout 4x5 MOVE (1 hour). Left plots show the average reward over the number of iterations, while right plots show the score (i.e., number of columns correctly broken) of the best policy computed so far (i.e., the results obtained in runs without exploration). The figures show how the agent is able to progressively learn how to progress over the states of the DFA corresponding to the $\text{LTL}_f/\text{LDL}_f$ specification. Similar results are obtained in different configurations (e.g., different

[6]Other state representations are also suitable to learn the task.

sizes of the bricks). reported in the columns is encoded with the first letter being either M for MOVE and F for FIRE actions available, and the second letter being either L for LOCAL and G for GLOBAL for the sensor modality.

Sapientino. SAPIENTINO Doc is an educational game for 5-8 y.o. children where a small mobile robot has to be programmed to visit specific cells in a 5x7 grid. Some cells contain concepts that must be matched by the children (e.g., a colored animal, a color, the first letter of the animal's name), while other cells are empty. The robot executes sequences of actions given in input by children with a keyboard on the robot's top side. During execution, the robot moves on the grid and executes an action (actually a *bip*) to announce that the current cell has been reached (this is called a *visit* of a cell). A pair of consecutive visits are correct when they refer to cells containing matching concepts. As in the real game, we consider a 5x7 grid with 7 triplets of colored cells, each triplet representing three matching concepts. *State representation* is defined by the following features: f_x, f_y, f_θ reporting the pose of the agent in the grid. In this scenario, we consider two different agents: OMNI: omni-directional movements (actions: up, down, left, right), DIFFERENTIAL: differential drive (actions: forward, backward, turn left, turn right). With this specification, the agent can just learn how to move in the grid, but it cannot match related concepts.

Restraining bolt. Consider now the specifications S2: *visit at least two cells of the same color for each color, in a given order among the colors* (the order of the colors is predefined: first C_1, then C_2, and so on) and S3: *visit all the triplets of each color, in a given order among the colors.* The following additional features are needed to express and evaluate the corresponding formula: f_b reporting that a *bip* action has just been executed and f_c reporting the color of the current cell.

The restraining specifications for these games can be expressed with LTL_f formulas. A fragment of LTL_f formula for the first game relative to the first color C_1 is

$$\begin{array}{l}\neg bip\,\mathcal{U}(\bigvee_{j=1,2,3} cell_{C_1,j} \wedge bip) \wedge \\ \bigwedge_{j=1,2,3} \Box(cell_{C_1,j} \wedge bip \rightarrow \bigcirc\Box(bip \rightarrow \neg cell_{C_1,j})) \wedge \\ \bigvee_{j=1,2,3} \Box(cell_{C_1,j} \wedge bip \rightarrow \bigcirc(\neg bip\,\mathcal{U} \bigvee_{k\neq j} cell_{C_1,k} \wedge bip)\end{array}$$

For other colors C_{i+1}, we use a similar formula, but requiring that $\bigvee_{j=1,2,3} cell_{C_i,j} \wedge bip$ has already been satisfied.

Two agents and two restraining bolts can be combined to form 4 different learning situations. We show only two of them. Rows 3 and 4 of Figure 9.3 show the agents' learning ability (score = 14 for S2 specs, score = 21 for the S3 specs). Similar results are obtained in different configurations.

Cocktail party. For a service robot involved in a cocktail party we consider a representation of the state in terms of robot's pose and objects' (drinks and snacks) and people's location. The agent can move in the environment, grasp and deliver items to people, and get a reward when a delivery task is completed. The robot has no sophisticated people perception capabilities, and no memory is available in the underlying MDP modeling the domain, so the robot cannot get information about individual people or remember who received what. The robot in this scenario will just learn how to bring one item to any person (choosing the shortest path).

Restraining bolt. Consider the following specification: *serve exactly one drink and one snack to every person, but do not serve alcoholic drinks to minors.* As in the previous examples, the restraining bolt works on separate features, namely identity, age and received items[7] and uses an $\text{LTL}_f/\text{LDL}_f$ formula to model this specification.

[7] In practice, services like Microsoft Cognitive Services Face API can be integrated into the bolt

operating scenario. We assume the map of the environment to be known, people sitting at tables in predefined known positions and locations of snack and drink items also known. From these information we can instantiate a simulator for the robot to navigate in this environment and reach the different locations[8].

For learning this task, we considered a problem with two people and two different kinds of drinks and snacks (4 tasks to be executed in total) and we implemented an abstract simulator reproducing the scenario of RoboCup@Home competition. The results of learning the restrained task in the simulator are depicted in Row 5 of Figure 9.3 (score = 4 means that the 2 persons have received one drink and one snack each). As shown, after about 1 minute of simulation[9], the RL agent converged to a policy satisfying the RB specifications.

Minecraft. As an example of our approach's modularity, we used the same agent of SAPIENTINO in a MINECRAFT scenario. Here the agent has to accomplish 10 tasks (described with non-Markovian rewards via an $\text{LTL}_f/\text{LDL}_f$ formula). The two agents share the same state representation S but differ in the action set A, the fluent configurations $\mathcal{L}$, and the component progressing the DFAs. Results (not shown here) confirm that a *general-purpose* agent can learn several tasks by only receiving information from its restraining bolt.

9.5 Summary and Discussion

In this chapter, we presented the novel concept of Restraining Bolts, where an authority imposes a restraining specifications to the learning agent, but they have different representations of the world. We formalized the problem in the case the agent is a Reinforcement Learning agent and the restraining specification is written in a $\text{LTL}_f/\text{LDL}_f$ formula φ, and showed that in fact the agent, despite ha has no access to the authority's world representation (i.e. the alphabet of φ), he can aim at optimizing the reward coming from the temporal logic specification, at the cost of handling an additional feature in its state space needed to keep track of the partial satisfaction of the temporal specification (i.e. the automaton state during the evalutation of the trace). However, there is no guarantee that the agent will actually able to satisfy it as it depends on the capabilities of the agent. We discussed possible improvements in terms of sampling efficiency by means of automatic reward shaping techniques, and provided several examples showing the relevance and the usefulness of the approach.

We have shown how to perform RL with $\text{LTL}_f/\text{LDL}_f$ restraining specifications by resorting to typical RL techniques based on MDPs. Notably, we have shown that the features needed to evaluate $\text{LTL}_f/\text{LDL}_f$ formulas can be kept separated from those directly accessible to the learning agent.

The work presented in this chapter can be ascribed to that part of research generated by the urgency of providing safety guarantees to AI techniques based on learning (Amodei et al., 2016; Hadfield-Menell et al., 2017; Orseau and Armstrong, 2016). In particular, it shares similarities with recent work on constraining the RL agent to satisfy certain safety conditions (Wen, Ehlers, and Topcu, 2015; Achiam et al., 2017; Alshiekh et al., 2018). There are however important differences. First, in enforcing the restraining bolt we consider the learning agent essentially as a black box. That is, the restraining bolt does not need to know the internals S of the

to provide this information.

[8]Specifically, we used Stage simulator in ROS with standard navigation stack.

[9]This time can be drastically reduced using optimized code.

learning agent, and specifies the desired constraints using only its world features $\mathcal{L}$. On the other hand, we do not guarantee the satisfaction of the restraining bolt constraints during training, as in (Achiam et al., 2017). In fact, differently from (Wen, Ehlers, and Topcu, 2015; Alshiekh et al., 2018), we do not guarantee the hard satisfaction of constraints even after training. After all "You can't teach pigs to fly"! and we may very well ask to do so in our restraining bolts, being these completely unrestricted in the selection of world features and in the kind of formulas they specify over such features. If we want to check formally that the optimal policy satisfies the restraining bolt specification, we first need to model how actions affect the restraining bolt's features $\mathcal{L}$, i.e., we need to link the learning agent's features S to $\mathcal{L}$, and then we can use, e.g., model checking. Notably, for doing RL we do not need to specify such a link, but we can simply allow the (possibly simulated) world to act as the link, in line with what advocated, e.g., in (Brooks, 1991), and very differently from what typically considered in knowledge representation (Reiter, 2001).

Clearly, the two separate representations (i.e., the two sets of features) need to be somehow correlated in reality. The crucial point, however, is that in order to perform RL effectively, *such a correlation does not need to be formalized*. In this work, we set this framework and provide proofs and experimental evidence that an learning agent can learn policies that optimize the conformance to the $\text{LTL}_f/\text{LDL}_f$ goals without including in the state space representation the features needed to evaluate the corresponding $\text{LTL}_f/\text{LDL}_f$ formula (more details can be found in (De Giacomo, Iocchi, et al., 2018).) Using these results, we can envision that once the agent is equipped with the restraining bolt, by simulating in its mind how to act (i.e., applying RL), it will deliberate a course of actions that automatically conform (as much as possible) to the restraining rules.

Apart from restraining bolts, the interest in having separate representations is manifold. The learning agent feature space can be designed separately from the features needed to express the goal, thus promoting *separation of concerns* which, in turn, facilitates the design, providing for *modularity* and *reuse* of representations (the same agent can learn from different bolts and the same bolt can be applied to different agents). Also, a reduced agent's feature space allows for realizing *simpler agents* (think, e.g., of a mobile robot platform, where one can avoid specific sensors and perception routines), while preserving the possibility of acting according to complex declarative specifications which cannot be represented in the agent's feature space. We plan to investigate this separation further in the future.

We now consider several directions for future work.

9.5.1 Learning $\text{LTL}_f/\text{LDL}_f$ goals

One interesting direction is to learn the $\text{LTL}_f/\text{LDL}_f$ goals. This is related to what in Business Process Management is called (declarative) *process mining* (Aalst, 2011; Pesic, Schonenberg, and Aalst, 2007), but also to so-called *model learning* (Angluin, 1987; Angluin, Eisenstat, and Fisman, 2015; Vaandrager, 2017). LTL_f has also been used to model advice to guide the exploration of the RL algorithm (Icarte, Klassen, et al., 2017). This is an interesting aspect that could be considered in our case as well.

9.5.2 POMDPs

Our approach for RL for $\text{LTL}_f/\text{LDL}_f$ goals with reduced features space is different from RL for Partially Observable Markov Decision Processes (POMDP) (Kaelbling,

Littman, and Cassandra, 1998), where the features for evaluation the LTL$_f$/LDL$_f$ formulas are not directly accessible, but can be probabilistically estimated through observations. Instead, in our work, such features remain visible but are simply not used by the RL agent. Obviously, devising effective techniques for RL on POMDPs with LTL$_f$/LDL$_f$ rewards, by exploiting the connection with automata, is of great interest.

9.5.3 Quantitative Interpretation of Temporal Formulas

Another interesting direction for future work is to consider goals specified in logics that have a quantitative interpretation of temporal formulas (Almagor, Boker, and Kupferman, 2016; Kupferman, 2016). These kind of logics have been used with success in the context of Model Predictive Control (Raman et al., 2014).

9.5.4 Automata-based Reward Shaping

In Section 9.3, we described a generic approach to apply reward shaping based on the automaton state of the DFA. In particular, the automata-based reward shaping scheme requires the specification of a potential function $\Phi : Q \to \mathbb{R}$ that will be used in potential-based reward shaping. As a future work, we could devise different strategies to compute such function, e.g.:

- give to a state q of the automaton $\mathcal{A}$ a potential $\Phi(q)$ corresponding to the minimum distance from an accepting state, possibly scaled by a constant factor c.
- the same as above, but consider the maximum (or worst-case) distance from any accepting state.
- instead of a constant leap between consecutive states in the same path toward an accepting state, we could consider a discount factor $\lambda \in (0, 1)$, where $\Phi(q) = \lambda^d$, where d is the distance from an accepting state.
- use value-iteration over the DFA as if it were an MDP, and let $\Phi = V^*$ (e.g. see (Camacho, Icarte, et al., 2019)).

Most importantly, these approaches are automatic, i.e. they only require in input the DFA of the temporal logic specification. Then, would be interesting to benchmark all of these alternatives on common RL benchmark environments.

Another direction is to design Φ in an environment-dependent way, and take into account the state $s \in S$ in the potential function $\Phi(s, q)$.

9.5.5 Restraining Bolts with Clocks

One of the strongest assumption we made is that the sampling of the features and of the fluents is simultaneous. This makes very hard to use the $\bigcirc$ operator in restraining bolts. Therefore, we could devise a new device, called *clock*, which gives the designer of the system more flexibility on how fluents should be sampled and can be seen by the restraining bolt. The clock is, in general, a function from a history of fluents to a boolean, which is true if the last fluent configuration can be seen by the restraining bolts. The clock itself could be specified by a temporal logic formula φ_{clock}.

More generally, as future work, we plan to study the theoretical properties for this separation to be effective, as well as to use this insight to facilitate the design of actual robots embedded in real environments.

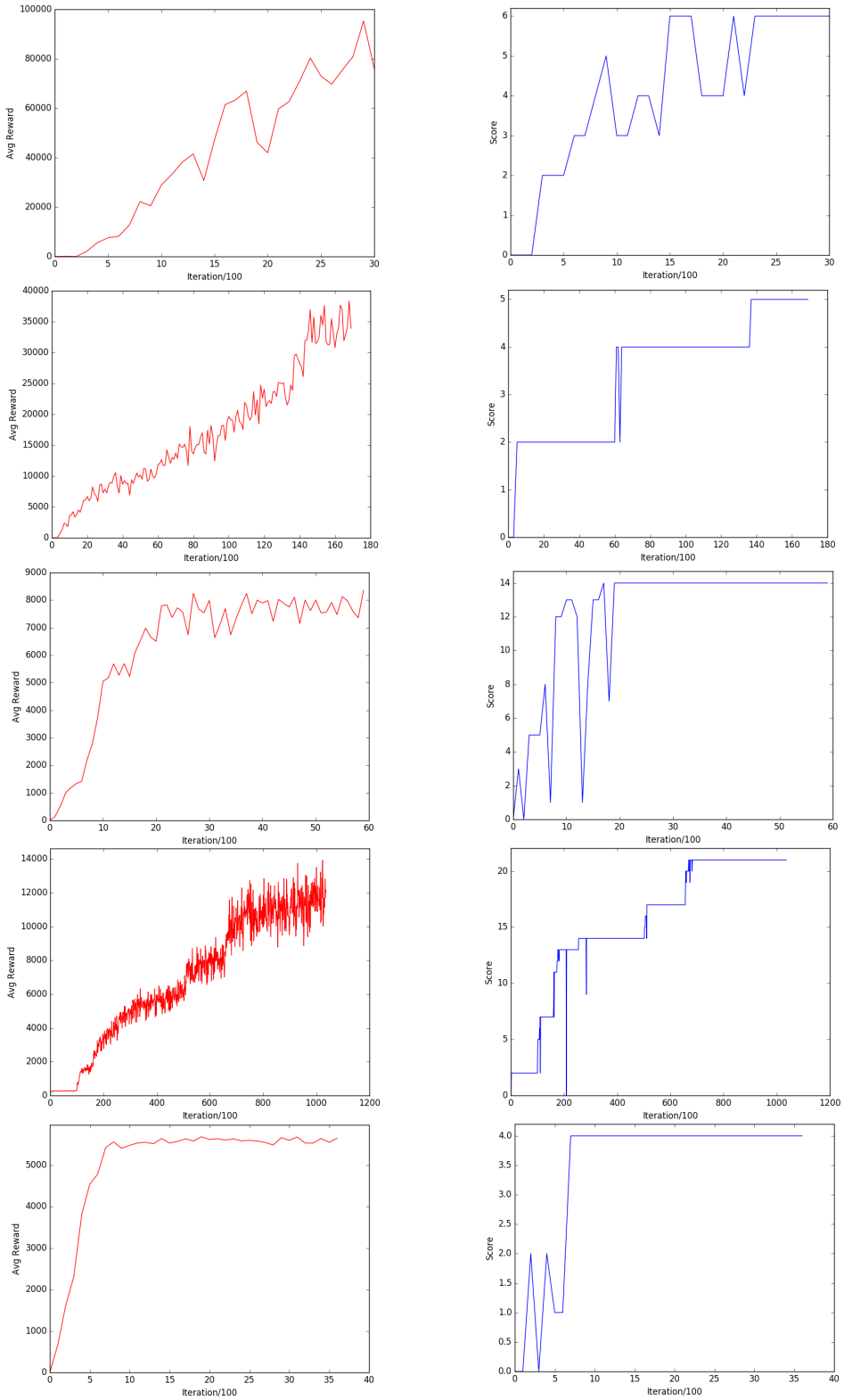


Figure 9.3. Average reward and scores over number of iterations. Row 1: Breakout MOVE + FIRE 4x6 bricks (5 minutes); Row 2: Breakout MOVE only 4x5 bricks (1 hour); Row 3: Sapientino S2 OMNI (3 minutes); Row 4: Sapientino S3 DIFFERENTIAL (1 hour); Row 5: Cocktail Party (3 minutes).

Chapter 10

Imitation Learning over Heterogeneous Agents

A common problem in Reinforcement Learning (RL) is that the reward function is hard to express. This can be overcome by resorting to Inverse Reinforcement Learning (IRL), which consists in first obtaining a reward function from a set of execution traces generated by an expert agent, and then making the learning agent learn the expert's behavior –this is known as Imitation Learning (IL). Typical IRL solutions rely on a numerical representation of the reward function, which raises problems related to the adopted optimization procedures.

We describe an IL method where the execution traces generated by the expert agent, possibly via planning, are used to produce a logical (as opposed to numerical) specification of the reward function, to be incorporated into a device known as *Restraining Bolt* (RB). The RB can be attached to the learning agent to drive the learning process and ultimately make it imitate the expert. We show that IL can be applied to *heterogeneous* agents, with the expert, the learner and the RB using different representations of the environment's actions and states, without specifying mappings among their representations.

The rest of the chapter is structured as follows:

- In Section 10.1, we introduce the problem of interest and the motivations.
- In Section 10.2, we consider related works.
- In Section 10.3, we formalize the problem using Restraining Bolts and model learning.
- In Section 10.4, we provide a solution using Angluin's L^* algorithm.
- In Section 10.5, we present several case studies.
- Section 10.6 concludes the chapter.

The contents of this chapter have been published in the conference paper (De Giacomo, Favorito, Iocchi, and Patrizi, 2020).

10.1 Introduction

Inverse Reinforcement Learning (IRL) consists in estimating a reward function from a set of traces captured during the execution of an agent's policy. IRL can be

used in many application domains to implement forms of Imitation Learning (IL). In IL, an expert agent executes a possibly optimal policy, generating a set of execution traces which are exploited by a learning agent to reconstruct the reward function, in turn used to learn a (possibly optimal) policy that imitates the expert behavior. Providing examples of the (optimal) policy is a very convenient way to specify goals for the learning agent, in contrast to defining a reward function, which is typically cumbersome. Interestingly, expert and learner may be different kinds of agents, e.g., human and robot, executing tasks in different ways, i.e., with different action and perception abilities. Unfortunately, this prevents an off-the-shelf application of the IRL approach as, in the classical IRL setting, the expert and the learner must share the representation space (e.g., states and actions).

In this chapter, expert and learner may have different capabilities representations of states and actions; thus, the learner cannot interpret the traces generated by the expert. To deal with this, we exploit the idea of Restraining Bolt (RB) (De Giacomo, Iocchi, et al., 2019): a device, with its own sensors, that can be attached to a Reinforcement Learning (RL) agent, to constrain its behavior and make it fulfill desired temporal high-level goals. Such goals are expressed as formulas of linear-time temporal logic over finite traces, LDL_f (De Giacomo and Vardi, 2013), over a set of fluents, generally different from the features used by the RL agent.

We consider a setting where the expert agent executes its policy, producing desired and undesired traces at its own representation level. From these traces, we obtain a deterministic finite-state automaton (DFA) accepting all the positive (desired) traces and rejecting all the negative (undesired) ones. The DFA thus represents (an approximation of) the expert's behavior. Then, based on a well-known equivalence between LDL_f and DFAs (De Giacomo and Vardi, 2013), the DFA is incorporated into a RB attached to the RL agent, to make it learn a (possibly optimal) policy that imitates the expert's behavior.

The ability of the RB to guide learning for a RL agent, when both use different representations and with no explicit mapping between them, has been proven in (De Giacomo, Iocchi, et al., 2019). Here, we exploit that result to implement an IL procedure where the specification of the transferred behavior is provided at a higher level wrt the states of the MDP. In other words, the low-level traces generated by the expert are transformed into high-level traces from the RB sensors. Once the high-level behavior is learned, this can be transferred to an agent with different capabilities. This process can be seen as IRL at the RB representation level: instead of estimating the reward function, we reconstruct the DFA associated with the goal formula and then use it for learning.

The main advantage of this setting is a higher modularity, as agents and RBs can be combined to form complex IL systems with minimal effort. Indeed, as discussed in (De Giacomo, Iocchi, et al., 2019), a RL agent can be extended to receive signals from a RB in a domain-independent way, by simply extending its state with an integer variable to store (an encoding of) the current state of the RB's DFA.

Summarizing, our contribution is an IL technique for agents with completely different state-action representations. The technique is based on the use of a RB to specify a high-level behavior and does not require any explicit mapping between the different representations.

10.2 Related work

Most IRL solutions model the reward function in parametric form (e.g., a weighted sum of reward features) and use some optimization or regression method

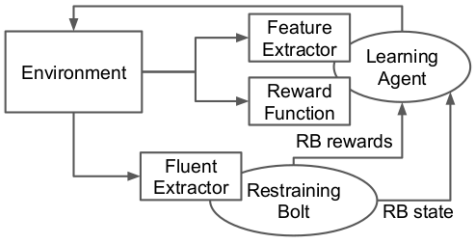


Figure 10.1. The RB setting

to optimize the parameter values (see (Arora and Doshi, 2018) for a recent survey of IRL solutions). The main issue with these approaches is that the optimization problem is essentially ill-posed, as many reward functions exist (including that with all null values) that can explain the observations, and defining metrics for their comparison is difficult (Ng and Russell, 2000). E.g., two reward functions differing only in one state-action pair may produce considerably different behaviors. Although these solutions solve several issues in IRL, estimating numerical reward functions from execution traces remains an open problem. This work aims, instead, at synthesizing the reward function at a logical level (the RB's representation level), avoiding numerical optimization and regression, thus overcoming their respective limitations. The proposed approach allows also for dealing with non-Markovian rewards.

Two broad classes of approaches, *passive* and *active*, exist to learn a (temporal) formula/DFA from sets of positive and negative traces. In passive approaches the formula/DFA is learned from a fixed set of positive and negative traces. Examples include (Camacho and McIlraith, 2019b) and (Heule and Verwer, 2010). In the former, an LTL_f formula satisfied by all positive traces and by no negative trace is generated. In the latter, the problem is compiled into SAT and then, by exploiting the SAT technology, a minimum-size DFA (wrt number of states) accepting all positive and no negative traces is obtained. In *active learning*, the set of traces is produced as the result of an interaction between the learner and the expert. The distinctive feature of this approaches is the fact that the expert knows the target formula/automaton (which is not the case in this work). Angluin's L^* algorithm (Angluin, 1987) (and later extensions) offers an example of this. While our work is agnostic to the learning technique, which is used in a black-box fashion, we resort to active learning, specifically Angluin's technique, using some care to overcome the fact that the expert does not know the target formula/automaton.

10.3 Problem definition

A Restraining Bolt (RB) is a tuple $RB = \langle \mathcal{L}, \{(\varphi_i, r_i)\}_{i=1}^{m} \rangle$ where each φ_i is an LDL_f formula over a set of fluents $\mathcal{L}$ and each r_i is a reward value. Fig. 10.1 illustrates the basic RB setting. This is a standard RL scenario, with the environment, the RL agent, its features and the reward function, extended with the RB, i.e., a device that observes the environment and, based on its own *fluents*, offers rewards to the agent. Fluents constitute the RB's representation of the environment state and need not match the RL agent features (and typically they do not). Formulas φ_i specify the behaviors that should be rewarded, each with its respective r_i. As known (De Giacomo and Vardi, 2013) LDL_f formulas can be equivalently represented as DFAs.

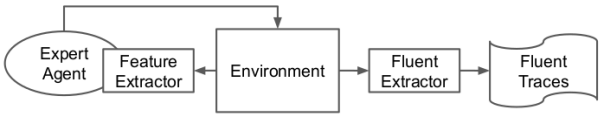


Figure 10.2. Trace generation for RB-IRL

We use this representation. RBs were introduced in (De Giacomo, Iocchi, et al., 2019), to constrain an agent's behavior to fulfill high-level (i.e., fluent-based) goals. We refer to that work for further details.

We use RBs to address the problem of transferring a task from an expert to a learner agent. The task is represented by a formula φ in a RB or, more precisely, by the corresponding DFA Q. As a result, we consider RBs of the form $\langle \mathcal{L}, Q, r \rangle$, where Q is a DFA representing an $\text{LTL}_f/\text{LDL}_f$ formula and r is a reward value associated with the accepting states of Q.

Consider now an expert agent defined on an MDP $\mathcal{M}_e = \langle S_e, A_e, Tr_e, R_e \rangle$. The agent can execute optimal policies of a given target task represented by a DFA Q, but cannot make the corresponding reward function explicit; in other words, the agent knows how to accomplish the task but cannot describe it. As the agent executes the policy, some traces are produced, some of which are desirable (posiive) and some other are not. The expert can correctly classify the traces as positive or negative, based on its own state representation.

On the other hand, the traces can also be seen from the RB perspective, through the RB sensors. Thus, from each state, the fluents can be extracted to produce the corresponding representation in the RB space. Notice the expert does not know anything about fluents, in particular, it cannot interpret them, as belonging to a different representation space. In fact, the expert is not even aware of the RB. This scenario is illustrated in Figure 10.2. Let $\mathcal{T}$ be a set of fluent traces collected while observing the behavior of the expert. The problem we address in this work is that of reconstructing a DFA $Q_\mathcal{T}$ that is consistent with $\mathcal{T}$, i.e., that accepts all of its positive traces and none of its negative. The approach proposed in this work allows for generating a new RB $RB = \langle \mathcal{L}, Q_\mathcal{T}, r \rangle$, where $Q_\mathcal{T}$ is the DFA built from $\mathcal{T}$, and r is a reward value associated with the accepting states of $Q_\mathcal{T}$. After the training phase, the generated RB can be placed on a learner agent to drive the learning process of a behaviour imitating the expert's.

Consider a learner agent defined on $\mathcal{M}_l = \langle S_l, A_l, Tr_l, R_l \rangle$, with Tr_l and R_l unknown, equipped with the RB that encodes the behavior of the expert agent in performing the given task. The system $M_l^{RB} = \langle M_l, RB \rangle$ can be used to learn an optimal policy driven by RB, as explained in (De Giacomo, Iocchi, et al., 2019). In this way, the behavior of the learner agent imitates that of the expert, when considering the evolution at the RB level.

Notice again that the state representations of S_e and S_l, as well as the set of actions A_e and A_l, may be completely different (e.g., states may come from different sets of state variables), as long as they allow to solve the task. Moreover, no explicit mapping between them is required.

10.4 Solution method

The core problem of our approach is extracting the DFA (or the formula) from the set $\mathcal{T}$ of (positive and negative) traces, to be incorporated in the RB used by

Figure 10.3. RB's DFA learning setting

the learning agent to learn the expert's behavior. This is illustrated in Fig. 10.3. Notice that the target DFA is unknown, even to the expert. As a result, the best we can hope for is to come up with an approximation. For this reason, we search for a DFA that accepts all positive and no negative traces, according to $\mathcal{T}$.

Since we aim at generalizing over the data $\mathcal{T}$, the obtained DFA must accept more traces than exactly the positive ones, and possibly reject more than the negative ones. As typical in learning, in order to guarantee a certain degree of generalization, some bias must be introduced. One reasonable approach is to check smaller DFAs (in terms of number of states) first. This is motivated by the intuition that smaller DFAs are less selective and tend to accept more traces than those with a large number of states. As a result, we expect them to generalize better than large ones.

As discussed, several approaches exist for extracting a DFA from a set of labelled traces. In our case, any is a reasonable candidate. For simplicity, we have selected L^* (Angluin, 1987). The choice was due to the following reasons. Firstly, the algorithm returns a DFA (not a formula) that can be used as-is when executing the RB –in the case of a formula, instead, this should be translated into its equivalent DFA representation first. Secondly, the algorithm produces increasingly larger DFAs, thus satisfying the generalization requirement discussed above –though it is not guaranteed to return the minimal DFA. Thirdly, the technique has been implemented in a reliable software framework, *LearnLib*, actively tested and maintained, which has proven extremely convenient for the implementation step[1]. We remark that our approach is agnostic to the specific DFA/formula extraction technique.

The algorithm works as follows. The learner poses *membership* queries ("is this a positive trace?") and *equivalence* queries ("is this the target formula/automaton?") to the expert, which answers respectively with a "yes/no" and a "yes" or a counterexample ("no, because this trace should be/not be accepted"). The learner uses membership queries to produce a candidate DFA. Once done with this, the learner asks the expert whether the candidate solution is the target DFA. If the expert answers "yes", the DFA has been found and no other work is required; if the expert answers "no", it also returns a counterexample which is used by the learner, together with possible additional membership queries, to produce a new candidate solution to be checked for equivalence, and so on. The algorithm is shown to terminate and find the target DFA in polynomial time wrt both the size of the minimal DFA equivalent to the target DFA and the maximum length of any returned counterexample (Angluin, 1987).

Unfortunately, in our case, the expert executes the policy offline, thus cannot be asked whether a certain trace is positive or negative, and, more critically, does not know the target DFA, thus cannot perform the equivalence check. As a consequence, the expert cannot act as the oracle required by L^* to answer the queries. Nonetheless, using the dataset $\mathcal{T}$, the oracle can be simulated in such a way that L^* generates a suitable approximation of the target DFA. This is done as follows: when a membership query is posed, the (simulated) oracle answers "no", if the input trace is classified as negative in $\mathcal{T}$, otherwise it answers "yes"; when an equivalence check is

[1] https://learnlib.de

to be performed, the oracle answers "yes " if the candidate DFA accepts all positive traces and no negative trace from $\mathcal{T}$, and "no" otherwise, returning also one of the traces that made the test fail.

As it can be easily seen, with this trick we can simulate the required oracle and thus apply the algorithm. Also, by the choices above, the returned DFA is an approximation of the (unknown) target DFA, in the sense discussed above. Notice that we are potentially accepting all traces that are not explicitly forbidden by $\mathcal{T}$. Obviously, this is not the only option and other are possible (e.g., classify randomly the traces not in $\mathcal{T}$), yet it is a possible way to achieve a generalization wrt the data set.

10.5 Case studies

We showcase our approach in three scenarios: Breakout (De Giacomo, Iocchi, et al., 2019), Sapientino (De Giacomo, Iocchi, et al., 2019) and Minecraft (Icarte, Klassen, et al., 2018b) (see Figure 10.4). For each scenario, we proceeded as follows.

Firstly, we fixed a restraining bolt in $\text{LTL}_f/\text{LDL}_f$, representing the target task, and we played a simplified version the game (that we call variant A), recording its traces (projected on fluents only) and labeling the good or bad according to the satisfaction of the formula. In this way, we generate an "expert behavior" that can be used later for assessing the quality of the policy learned. Then, we used the collected traces to generate a DFA that captures the expert's behavior, using the *LearnLib* implementation of Angluin's algorithm, as described in the previous section. Such a DFA typically is not the same as the $\text{LTL}_f/\text{LDL}_f$ formula but it is close enough. Next, the learner learns a policy in the more complex game (variant B) using the learned DFA as the restraining bolt, using the same approach described in (De Giacomo, Iocchi, et al., 2019). Finally, to assess policy learned we simulate its execution together with the original $\text{LTL}_f/\text{LDL}_f$ DFA checking when we reach its final states.

Notice that for each scenario, the features and the actions in variant A and variant B are different. In particular, in variant A actions are stronger making the game easier. Notice that there must be a relationship (but not an isomorphism) between the actions in the two variants for making the approach effective in practice, but such a relationship can be quite loose and can remain unexpressed. Note also that actions are not used in the alphabet to progress the DFA and hence are not part of the reward given by the restraining bolt. This allows us to have different actions in the two variants. What is crucial for our approach to work is to have enough good and bad high-level traces to learn an accurate DFA. Once we get DFA, we assign a reward to the final states, and apply the restraining bolt techniques (De Giacomo, Iocchi, et al., 2019).

We next describe each, scenario together with the corresponding variants and the target task. In all cases, the generated DFA was consistent with the target task and the learner was able to learn the task. The code can be found at `https://github.com/whitemech/Imitation-Learning-over-Heterogeneous-Agents-with-Restraining-Bolts`.

Breakout. This is the popular Atari game where a brick wall must be destroyed. In the original version, bricks can be removed by hitting them with a ball driven by a paddle placed at the bottom of the screen, that can move only horizontally. As **variant** A, we considered the game where there is no ball and the paddle can fire bullets to break the bricks. In this case, the state representation of the expert consists in the paddle position only. Notice that the brick configuration is not accessible to the agent (and neither is, consequently, the configuration of columns).

Figure 10.4. Experimental scenarios: BREAKOUT, SAPIENTINO, MINECRAFT

As **variant** B, instead, we used the original version. In this variant, the learner can hit the ball with the paddle (but cannot fire) and can access the ball velocity and position. The **target task** is: *all bricks must be removed, completing the columns from left to right*, i.e., all the bricks in column i must be removed before completing any other column $j > i$. This task can be expressed with an LDL_f formula, using a fluent to represent the state of each column (completed or not).

Sapientino. SAPIENTINO is an educational game for 5-8 y.o. children where a small mobile robot must be programmed to visit specific cells in a 5x7 grid. Some cells contain concepts that must be matched by the children (e.g., a colored animal, a color, the first letter of the animal's name), while other cells are empty. The robot executes sequences of actions given in input by children with a keyboard on the robot's top side. During execution, the robot moves on the grid and executes an action (actually a *beep*) to announce that the current cell has been reached (this is called a *visit* of a cell).

As **variant** A, we took the scenario where the (expert) robot can move omni-directionally (actions: up, down, left, right). As **variant** B, we took the scenario where the (learner) robot can move differentially (actions: forward, backward, turn left, turn right). In both variants, the robot cannot see its position on the grid, nor can sense colors and/or concepts on the cells. The target task is: *visit a sequence of colors in a given order without bad beeps between the visits.*

Minecraft. In this scenario, an agent has to accomplish a task consisting in a sequence of *get resource* and *use tool* actions. A requirement to get resources and use tools is that the robot be on the cell associated with the resource or tool.

In **variant** A, the expert is endowed with "teleporting" capabilities (e.g. "go to a resource/tool"). In **variant** B, the learner can move only on the grid using differential drive, similarly variant B of SAPIENTINO.

Results of learning variant B tasks are similar to the ones presented in (De Giacomo, Iocchi, et al., 2019) and are thus not reported here.

10.6 Summary and Discussion

We have shown an approach based on the use of Restraining Bolts to perform Imitation Learning in a scenario where *heterogeneous* agents are involved, i.e., where, in particular, the expert and the agent do not share the same state representation nor a mapping between them. We do so by applying an approach based on Inverse Reinforcement Learning, where the behavior of the expert agent is learned and represented as a DFA that is then incorporated in a RB, in turn used by the learner at training time. Interestingly, the DFA constitutes a logical representation of the reward function, thus avoiding a number of problems arising when a numerical representation is adopted. We have performed experiments on several use-cases to show the effectiveness of our approach. Despite the differences in the state-action representation space, in all cases our approach was successful in transferring a task from the expert to the learner.

Future directions include testing different approaches to the generation of the DFA from a set of traces, as well as other approximation criteria.

Chapter 11

Temporal Logic Monitoring Rewards via Transducers

In Markov Decision Processes (MDPs), rewards are assigned according to a function of the last state and action. This is often limiting, when the considered domain is not naturally Markovian, but becomes so after careful engineering of an extended state space. The extended states record information from the past that is sufficient to assign rewards by looking just at the last state and action. Non-Markovian Reward Decision Processes (NMRDPs) extend MDPs by allowing for non-Markovian rewards, which depend on the history of states and actions. Non-Markovian rewards can be specified in temporal logics on finite traces such as $\text{LTL}_f/\text{LDL}_f$, with the great advantage of a higher abstraction and succinctness; they can then be automatically compiled into an MDP with an extended state space. In this chapter, we contribute to the techniques to handle temporal rewards and to the solutions to engineer them. The chapter is structured as follows:

- Section 11.1 provides the context of this work and the motivations.
- Section 11.2 provides background knowledge: transducers, NMRDPs and Runtime Monitoring[1].
- In Section 11.3, we formally define reward transducers, and the operations of *sum* and *direct sum* over them. In particular, we show how to compile temporal rewards which merges the formula automata into a single transducer, sometimes saving up to an exponential number of states.
- In Section 11.4, we extend MDPs using reward transducers; the reward transducers fully represent the reward function of the MDP.
- In Section 11.5, we show how temporal specifications can capture any Markovian reward function, and reward functions expressed by reward transducers.
- In Section 11.6, we introduce a novel type of temporal logic specification, which draws inspiration from the runtime monitoring community. We define monitoring rewards, which add a further level of abstraction to temporal rewards by adopting the four-valued conditions of runtime monitoring; we argue that our compilation technique allows for an efficient handling of monitoring rewards.

[1]Some topics, although already present elsewhere in the thesis, are reproposed as they give the notation for the rest of the chapter.

- In Section 11.7, we discuss some applications of monitoring rewards in the context of Reinforcement Learning classical problems.
- Section 11.8 concludes the chapter.

The contents of this chapter have been published in the conference paper (De Giacomo, De Masellis, et al., 2020).

11.1 Introduction

In a Markov Decision Process (MDP) (Puterman, 1994) the transition probability function and the reward function are Markovian, i.e., they depend only on the last state and action. However, this limitation does not allow for rewarding behaviours that extend overtime; alternatively, it requires to engineer an extended state space where states record enough information from the past. To overcome such limitations, non-Markovian Reward Decision Processed (NMRDP) have been proposed (Bacchus, Boutilier, and Grove, 1996; Thiébaux et al., 2006). In particular, the idea is to encode non-Markovian rewards into an MDP by extending the state space, with minimality guarantees of the resulting MDP.

The same idea, with some variations, has been investigated in more recent works. In (Icarte, Klassen, et al., 2018a; Camacho, Icarte, et al., 2019; Icarte, Waldie, et al., 2019), the authors introduce the concept of *reward machine*, an automata-based formalism to encode non-Markovian rewards. In (Quint et al., 2019), formal languages are used to specify soft and hard constraints on actions, by enforcing constraints on the action space, called *action shaping*. In (Alshiekh et al., 2018), an approach based on temporal logic has been used to monitor the actions of an agent and to prevent the violation of critical safety specifications. In (Brafman, De Giacomo, and Patrizi, 2018; De Giacomo, Iocchi, et al., 2019), rewards are specified in the temporal logics $\text{LTL}_f/\text{LDL}_f$ (De Giacomo and Vardi, 2013; De Giacomo and Vardi, 2015; De Giacomo and Vardi, 2016). Here the construction of the extended MDP is based on the correspondence between such logics and finite-state automata (Rabin and Scott, 1959). Specifically, the extended MDP is obtained as the synchronous product of a formula's automata with the automata underlying the NMRDP. All of these techniques are examples of how much Knowledge Representation can be of great help for reward specification.

A crucial property of such techniques is the *overhead* required to handle the non-Markovianity. Such overhead is introduced in the original state space to generate the extended MDP over which the learning is performed. It is desirable that the overhead is the minimum possible since it affects the effectiveness of learning algorithms (e.g. the exploration phase in Reinforcement Learning (Sutton and Barto, 1998)).

In this work, we want to extend the approach in (Brafman, De Giacomo, and Patrizi, 2018) while keeping such overhead to the minimum. To do so, we merge automata of the various formulas used for the rewards into a single *transducer* from traces to rewards (i.e. outputs a reward for every prefix of the trace), which encodes all the temporal specifications in a single finite-state machine. This gives us further opportunities of minimizations if we do not care from the satisfaction of which formula a given reward is obtained. Indeed, we show that by giving up this information, the transducer can be exponentially (in fact factorially) smaller than the minimal automaton in (Brafman, De Giacomo, and Patrizi, 2018), and never worse in general. Then, inspired by the literature on *monitoring* (Bauer, Leucker, and Schallhart, 2010; Ly et al., 2013; De Giacomo, Masellis, Grasso, et al., 2014), we devise a way of specifying rewards using $\text{LTL}_f/\text{LDL}_f$ which associates reward

not to simply the satisfaction of the formula, but to the four classical monitoring conditions: the formula is temporarily true, temporarily false, permanently true, and permanently false. We illustrate the convenience of this kind of $\text{LTL}_f/\text{LDL}_f$ reward specifications and show that these four conditions can be monitored at no additional cost w.r.t. to satisfaction only, through the use of transducers. Finally, we discuss the use of such kind of $\text{LTL}_f/\text{LDL}_f$-based reward specifications in reinforcement learning of non-Markovian specifications.

11.2 Background

MDPs and RL. A Markov Decision Process (MDP) $\mathcal{M} = \langle S, A, Tr, R\rangle$ contains a set S of states, a set A of actions, a transition function $Tr : S \times A \to Prob(S)$ that returns for every state s and action a a distribution over the next state, and a reward function $R : S \times A \to \mathbb{R}$ that specifies the reward (a real value) received by the agent when transitioning from state s to state s' by applying action a. We see states S as truth assignments to a set $\mathcal{P}$ of propositional atoms. A solution to an MDP is a function, called a *policy*, assigning an action to each state, possibly with a dependency on past states and actions. The *value* of a policy ρ at state s, denoted $v^\rho(s)$, is the expected sum of (possibly discounted by a factor γ, with $0 \leq \gamma \leq 1$) rewards when starting at state s and selecting actions based on ρ. Typically, the MDP is assumed to start in an initial state s_0, so policy optimality is evaluated w.r.t. $v^\rho(s_0)$. Every MDP has an *optimal* policy ρ^*. In discounted cumulative settings, there exists an optimal policy that is *Markovian* $\rho : S \to A$, i.e., ρ depends only on the current state, and deterministic (Puterman, 1994).

Reinforcement Learning (RL) is the task of learning a possibly optimal policy, from an initial state s_0, on an MDP where only S and A are known, while Tr and R are not—see, e.g., (Sutton and Barto, 1998).

Automata. A *(finite-state) automaton* is a computational model with limited capabilities. It can read input strings in a given alphabet, it keeps track of its current state among finitely many, and it can produce output strings. An automaton whose output response is limited to a simple 'yes' or 'no' is called an *acceptor*. A more general automaton, capable of producing strings of symbols as output, is called a *transducer*.

The most basic kind of automata are deterministic finite automata (DFA) (Rabin and Scott, 1959). A DFA is a 5-tuple $\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta\rangle$ where Q is the (non-empty) finite set of states, Σ is the set of input symbols, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times \Sigma \to Q$ is the transition function. The *extended transition function* δ^* of $\mathcal{A}$ is $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$. Automaton $\mathcal{A}$ *accepts* a word w if $\delta^*(q_0, w) \in F$. The *language of* $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts.

Two fundamental kinds of transducers are *Moore machines* and *Mealy machines*. A Moore machine (Moore, 1956) is a tuple $\mathcal{M}_o = \langle Q, \Sigma, \Gamma, q_0, \delta, \theta\rangle$ where Q is the set of states, Σ is the set of the input symbols, Γ is the set of the output symbols, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\theta : Q \to \Gamma$ is the output function that maps states to output symbols. The *output* of $\mathcal{M}_o$ on word $a_1 \dots a_n$ is $\theta(q_0)\,\theta(\delta^*(q_0, a_1)) \dots \theta(\delta^*(q_0, a_1, \dots, a_n))$. A Mealy machine $\mathcal{M}_e$ (Mealy, 1955) is like a Moore machine except that its output function $\theta : Q \times \Sigma \to \Gamma$ maps transitions to output symbols, instead of states. Hence the output of $\mathcal{M}_e$ on word $a_1 \dots a_n$ is $\theta(q_0, a_1)\,\theta(\delta^*(q_0, a_1), a_2) \dots \theta(\delta^*(q_0, a_1, \dots, a_{n-1}), a_n)$. Note that, for a Moore machine and a Mealy machine performing the same number of transitions,

the output of a Mealy has one symbol less. A Moore/Mealy machine $\mathcal{M}$ corresponds to the *transduction function* $\mathcal{F}_{\mathcal{M}} : \Sigma^* \to \Gamma^*$ that maps its input to its output. That is, such machines translate words on the input alphabet Σ to words on the output alphabet Γ. It can be shown that Moore machines and Mealy machines have the same expressivity, that is, for a Moore machine there exist an equivalent Mealy machine, and vice versa (Linz, 2006).

An important property that we will use in the next sections is that both DFAs and Moore/Mealy machines can be *minimised*, and the resulting minimal automata are unique (modulo state renaming) for the language they recognise or the transduction function they represent, respectively.

NMRDPs. A non-Markovian reward decision process (NMRDP) (Bacchus, Boutilier, and Grove, 1996) is a tuple $\langle S, A, Tr, \bar{R}\rangle$, where S, A and Tr are as in an MDP (with each in S being an assignment for propositions $\mathcal{P}$), but the reward $\bar{R}$ is a real-valued function over finite state-action sequences (referred to as *traces*), i.e., $\bar{R} : (S \times A)^* \to \mathbb{R}$. Given a (possibly infinite) trace $\pi = \langle s_0, a_1, \ldots, s_{n-1}, a_n\rangle$, the *value* of π is: $v(\pi) = \sum_{i=1}^{|\pi|} \bar{R}(\langle \pi(1), \pi(2), \ldots, \pi(i)\rangle)$, where $\pi(i)$ denotes the pair (s_{i-1}, a_i). In NMRDPs, policies are also non-Markovian $\bar{\rho} : S^* \to A$. Since every policy induces a distribution over the set of possible infinite traces, we can define the value of a policy $\bar{\rho}$, given an initial state s, as: $v^{\bar{\rho}}(s) = E_{\pi \sim M,\bar{\rho},s} v(\pi)$. That is, $v^{\bar{\rho}}(s)$ is the expected value of infinite traces, where the distribution over traces is defined by the initial state s, the transition function Tr, and the policy $\bar{\rho}$.

Specifying a non-Markovian reward function explicitly is cumbersome and unintuitive, even if only a finite number of traces are to be rewarded. $\text{LTL}_f/\text{LDL}_f$ provides an intuitive and convenient language for non-Markovian rewards (Camacho, Chen, et al., 2017b; Brafman, De Giacomo, and Patrizi, 2018). Following (Brafman, De Giacomo, and Patrizi, 2018) we can specify $\bar{R}$ using a set of pairs $\{(\varphi_i, r_i)\}_{i=1}^m$, where each φ_i is an $\text{LTL}_f/\text{LDL}_f$ formula over the propositions $\mathcal{P}$ that selects the traces to reward, and r_i the reward assigned to those traces. When the current (partial) trace is $\pi = \langle s_0, a_1, \ldots, s_{n-1}, a_n\rangle$, the agent receives at s_n each reward r_i whose formula φ_i is satisfied by π.

From NMRDPs to MDPs. In (Brafman, De Giacomo, and Patrizi, 2018) it is shown that for any NMRDP $M = \langle S, A, Tr, \{(\varphi_i, r_i)\}_{i=1}^m\rangle$, with φ_i being $\text{LTL}_f/\text{LDL}_f$ formulas, there exists an MDP $M' = \langle S', A, Tr', R'\rangle$ that is *equivalent* to M in the sense that the states of M can be (injectively) mapped into those of M' in such a way that corresponding (under the mapping) states yield the same transition probabilities, and corresponding traces have the same rewards (Bacchus, Boutilier, and Grove, 1996). Denoting with $\mathcal{A}_{\varphi_i} = \langle Q_i, 2^{\mathcal{P}} \times A, q_{i0}, \delta_i, F_i\rangle$ (notice that $S \subseteq 2^{\mathcal{P}}$ and δ_i is total) the DFA associated with φ_i, the equivalent MDP M' is as follows:

- $S' = Q_1 \times \cdots \times Q_m \times S$;
- $Tr' : S' \times A \times S' \to [0, 1]$ is defined as:

$$Tr'(q_1, \ldots, q_m, s, a, q'_1, \ldots, q'_m, s') = \begin{cases} Tr(s, a, s') & \text{if } \forall i : \delta_i(q_i, (s, a)) = q'_i \\ 0 & \text{otherwise;} \end{cases}$$

- $R' : S' \times A \to \mathbb{R}$ is defined as:

$$R'(q_1, \ldots, q_m, s, a) = \sum_{i:\delta_i(q_i,(s,a))\in F_i} r_i$$

Observe that the state space of M' is the product of the state spaces of M and $\mathcal{A}_{\varphi_i}$, and that the reward R' is Markovian. In other words, the (stateful) structure of the $\text{LTL}_f/\text{LDL}_f$ formulas φ_i used in the (non-Markovian) reward of M is *compiled* into the states of M'.

Theorem 11.1 ((Brafman, De Giacomo, and Patrizi, 2018)). *The NMRDP $M = \langle S, A, Tr, \{(\varphi_i, r_i)\}_{i=1}^m\rangle$ is equivalent to the MDP $M' = \langle S', A, Tr', R'\rangle$ defined above.*

Actually this theorem can be refined into a stronger lemma. A policy ρ for an NMRDP M and a policy ρ' for an equivalent MDP M' are *equivalent* if they *guarantee the same rewards*. Assume M' is constructed as above and let ρ' be a policy for M'. Consider a trace $\pi = \langle s_0, a_1, s_1, \ldots, s_{n-1}, a_n\rangle$ of M and assume it leads to state s_n. Further, let q_n^i be the state of $\mathcal{A}_{\varphi_i}$ on input π. We define the (non-Markovian) policy $\bar{\rho}$ equivalent to ρ' as $\bar{\rho}(s_0, \ldots, s_n) = \rho'(q_n^1, \ldots, q_n^m, s_n)$. Similarly, given a policy ρ for M, by just tracking the state of the DFAs $\mathcal{A}_{\varphi_i}$, it is immediate to define the equivalent policy ρ' for M'. Hence we have:

Lemma 11.2 ((Brafman, De Giacomo, and Patrizi, 2018)). *Given an NMRDP M and an equivalent MDP M', every policy ρ' for M' has an equivalent policy $\bar{\rho}$ for M and vice versa.*

Moreover, as observed by (De Giacomo, Iocchi, et al., 2019), it is possible to do RL over the M' equivalent to M. Being M' an MDP, this can be done by off-the-shelf RL algorithms (e.g., Q-learning and SARSA). Of course, neither M nor M' are (completely) known to the learning agent, and the transformation is never done explicitly. Rather, during the learning process, the agent assumes that the underlying model has the form of M' instead of that of M.

Theorem 11.3 ((De Giacomo, Iocchi, et al., 2019)). *RL for $\text{LTL}_f/\text{LDL}_f$ rewards over an NMRDP $M = \langle S, A, Tr, \{(\varphi_i, r_i)\}_{i=1}^m\rangle$, with Tr and $\{(\varphi_i, r_i)\}_{i=1}^m$ hidden to the learning agent can be reduced to RL over the MDP $M' = \langle S', A, Tr', R'\rangle$ defined above, with Tr' and R' hidden to the learning agent.*

Runtime Monitoring. We will borrow the four-valued semantics of runtime monitoring on finite traces (Bauer, Leucker, and Schallhart, 2010; Ly et al., 2013; De Giacomo, Masellis, Grasso, et al., 2014). Given an $\text{LTL}_f/\text{LDL}_f$ formula φ and a trace π, we say that:

- φ is *temporarily true* in π if π satisfies φ and there is a continuation of π that does not satisfy φ, and we write $\pi \models [\![\varphi = \mathit{temp\_true}]\!]$;
- φ is *temporarily false* in π if π does not satisfy φ and there is a continuation of π that satisfies φ, and we write $\pi \models [\![\varphi = \mathit{temp\_false}]\!]$;
- φ is *permanently true* in π if π and all its continuations satisfy φ, and we write $\pi \models [\![\varphi = \mathit{perm\_true}]\!]$;
- φ is *permanently false* in π if π and all its continuations do not satisfy φ, and we write $\pi \models [\![\varphi = \mathit{perm\_false}]\!]$.

11.3 Reward Transducers

In the literature, several approaches have been proposed to automatically extend MDPs so to capture non-Markovian or temporally-extended rewards. The central idea is to extend states with sufficient information from the past. In (Bacchus, Boutilier, and Grove, 1996; Thiébaux et al., 2006) rewards are expressed by using variants of LTL, and states are extended by suitably annotating them exploiting the structure of the reward formulas. In (Littman, 2015b; Littman et al., 2017) the necessity of a declarative mechanism for expressing complex rewards was again brought about to tame the difficulty of reward engineering in complex systems. In (Brafman, De Giacomo, and Patrizi, 2018) rewards where expressed using $\text{LTL}_f/\text{LDL}_f$ temporal logic on finite traces, which are first translated into DFAs, and then the extended MDP is obtained as the cross product of such DFAs with the original MDP—as discussed in Chapter 8. This approach is applied to reinforcement learning in (De Giacomo, Iocchi, et al., 2019).

In (Icarte, Klassen, et al., 2018a; Camacho, Icarte, et al., 2019) the idea of handling non-Markovian rewards through a finite machine (as a DFA) is decoupled from where the machine comes from (e.g., from an LTL_f specification) and the focus becomes the machine itself, called "reward machine".

In this work we first focus on reward machines directly as in (Icarte, Klassen, et al., 2018a; Camacho, Icarte, et al., 2019), introducing the general notion of *reward transducers*; then later we will show some advanced way of declaratively specifying such transducers that extend the ideas in (Bacchus, Boutilier, and Grove, 1996; Thiébaux et al., 2006; Brafman, De Giacomo, and Patrizi, 2018).

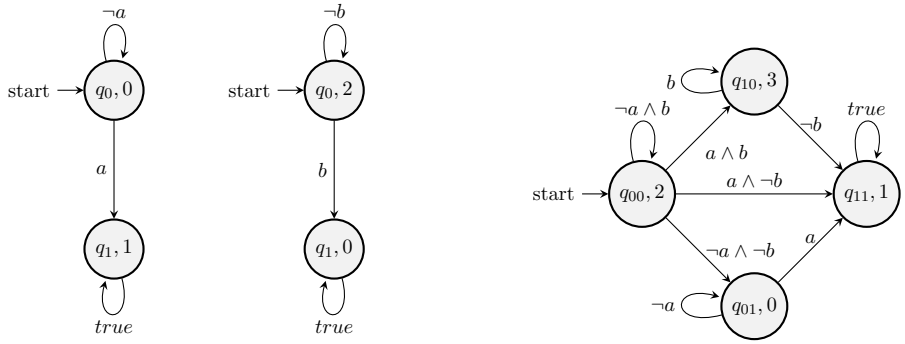
In our context, a reward transducer maps MDP traces of the form $\pi = \langle (s_0, a_1), (s_1, a_2), \ldots, (s_{n-1}, a_n) \rangle$ to sequences of rewards $r_1, r_2, \ldots, r_n$ (i.e. it outputs a reward for every prefix of the trace).

Definition 11.4. *For states S and actions A, a* reward transducer *is a transducer with input alphabet $S \times A$ and output alphabet $\mathcal{R} \subset \mathbb{R}$. A* Moore (or Mealy) reward machine *is a reward transducer that is a Moore (Mealy) machine.*

Thus, we can define any non-Markovian reward function $\bar{R}$ as a transduction function and we can implement it as a transducer. We observe that a temporal specification (φ, r) can be transformed into an equivalent reward transducer. Indeed, we can transform the associated DFA $\mathcal{A}_\varphi$ into the Moore machine $\mathcal{M}_o^\varphi = \langle Q, 2^\mathcal{P} \times A, \{0, r\}, q_0, \delta, \theta_o \rangle$ where Q, q_0 and δ are defined as in $\mathcal{A}_\varphi$, and $\theta_o(q) = 0$ if $q \notin F$, $\theta_o(q) = r$ otherwise. That is, the Moore machine outputs 0 for every prefix that does not satisfy φ and outputs r for every prefix that satisfies it. Analogously we can define a Mealy machine $\mathcal{M}_e^\varphi = \langle Q, 2^\mathcal{P} \times A, \{0, r\}, q_0, \delta, \theta_e \rangle$ where everything is defined like in the Moore machine but $\theta_e(q, (s, a)) = r$ if $\delta(q, (s, a)) \in F$, and 0 otherwise.

Sum of Reward Transducers. We now define the *sum* of two Moore reward machines $\mathcal{M}_o^1 = \langle Q_1, \Sigma, \mathcal{R}_1, q_{10}, \delta_1, \theta_1 \rangle$ and $\mathcal{M}_o^2 = \langle Q_2, \Sigma, \mathcal{R}_2, q_{20}, \delta_2, \theta_2 \rangle$, which is a Moore reward machine outputting the sum of the rewards of the two initial machines. Formally, the sum machine $\mathcal{M}'_o = \mathcal{M}_o^1 + \mathcal{M}_o^2$ is $\langle Q, \Sigma, \mathcal{R}, q_0, \delta, \theta_o \rangle$ where:

- $Q = Q_1 \times Q_2$;
- $\mathcal{R} = \{r_1 + r_2 \mid r_1 \in \mathcal{R}_1,\ r_2 \in \mathcal{R}_2\}$;
- $q_0 = (q_{10}, q_{20})$;



(a) $\mathcal{M}_o^1$ **(b)** $\mathcal{M}_o^2$ **(c)** $\mathcal{M}'_o = \mathcal{M}_o^1 + \mathcal{M}_o^2$

Figure 11.1. In Figure (a), the Moore reward machine $\mathcal{M}_o^1$ for the temporal specification $(\Diamond a, +1)$. In Figure (b), Moore reward machine $\mathcal{M}_o^2$ for the temporal specification $(\Box b, +2)$. In Figure (c), the Moore machine of the sum of $\mathcal{M}_o^1$ and $\mathcal{M}_o^2$.

- $\delta = \{(q_1, q_2) \xrightarrow{\sigma} (q'_1, q'_2) \mid \forall i \in \{1, 2\}.\ q_i \xrightarrow{\sigma} q'_i \in \delta_i\}$;
- $\theta_o = \{(q_1, q_2) \mapsto r_1 + r_2 \mid \forall i \in \{1, 2\}.\ \theta_i(q_i) = r_i\}$.

It is a standard cross-product construction, where the output of each new state is the sum of the outputs of the corresponding old states. As a result of the construction, the transduction function implemented by $\mathcal{M}_o^1 + \mathcal{M}_o^2$ is the sum of the functions of $\mathcal{M}_o^1$ and $\mathcal{M}_o^2$, i.e., $\mathcal{F}_{\mathcal{M}_o^1 + \mathcal{M}_o^2} = \mathcal{F}_{\mathcal{M}_o^1} + \mathcal{F}_{\mathcal{M}_o^2}$. The sum of two Mealy reward machines is defined very similarly to the sum of two Moore reward machines. The only thing that changes is how the output function is built:

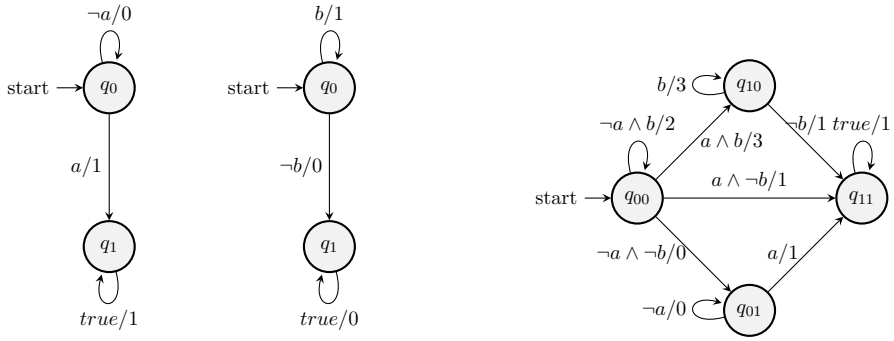
$$\theta_e = \{((q_1, q_2), \sigma) \mapsto r_1 + r_2 \mid \forall i \in \{1, 2\}.\ \theta_i(q_i, \sigma) = r_i\}.$$

Direct Sum of Reward Transducers. If the two machines are basically the same machine except for the output function, then we can build a sum machine simply by taking the sum of their output function. In this case we call the two machines *shape-equivalent*—a notion inspired by the shape-equivalence for DFAs in (De Giacomo, De Masellis, et al., 2020). Specifically, $\mathcal{M}_o^1$ and $\mathcal{M}_o^2$ are shape-equivalent if differ only in their output, or in other words have the same states, input alphabet, initial state, and transition function. For such machines, we can then define the *direct sum* machine $\mathcal{M}_o^1 \oplus \mathcal{M}_o^2 = \langle Q, \Sigma, \mathcal{R}, q_0, \delta, \theta \rangle$ where Q, Σ, q_0, and δ are the common states, input alphabet, initial state, and transition function, respectively, $\mathcal{R}$ is defined as for $\mathcal{M}_o^1 \oplus \mathcal{M}_o^2$, and $\theta = \theta_1 + \theta_2$. It is again the case that $\mathcal{F}_{\mathcal{M}_o^1 \oplus \mathcal{M}_o^2} = \mathcal{F}_{\mathcal{M}_o^1} + \mathcal{F}_{\mathcal{M}_o^2}$. Whenever in the following we take the sum of two machines, we can instead take their direct sum if we know that they are shape-equivalent. The same definition applies to the Mealy reward machines, except that the transition function depends on a state-symbol pair, rather than just a state.

In Figure 11.1, we show the Moore reward machines for the temporal specifications $(\Diamond a, +1)$ and $(\Box b, +2)$ and their sum. In Figure 11.2, we show the equivalent Mealy reward machines.

11.4 Extending MDPs via Reward Transducers

Rewarding complex behaviours is a challenging task, and temporal logic provides the right level of abstraction to address the problem (Littman, 2015b; Littman et al.,



(a) $\mathcal{M}_e^1$ **(b)** $\mathcal{M}_e^2$ **(c)** $\mathcal{M}'_e = \mathcal{M}_e^1 + \mathcal{M}_e^2$

Figure 11.2. Here we show the same specifications depicted in Figure11.1, but implemented as Mealy reward machines. In Figure (a), the Mealy reward machine $\mathcal{M}_e^1$ for the temporal specification $(\Diamond a, +1)$. In Figure (b), Moore reward machine $\mathcal{M}_e^2$ for the temporal specification $(\Box b, +2)$. In Figure (c), the Mealy machine of the sum of $\mathcal{M}_e^1$ and $\mathcal{M}_e^2$.

2017). This is the philosophy behind NMRDPs with $\text{LTL}_f/\text{LDL}_f$ rewards (Brafman, De Giacomo, and Patrizi, 2018). NMRDPs can be reduced to MDPs, and hence solved using off-the-shelf algorithms for MDPs. This comes, however, at the cost of an *extension* of the state space, which is required to keep track of the state of partial satisfaction of the temporal rewards. Such an extension introduces an *overhead* which is necessary to deal with non-Markovianity, but it is a computational cost for the algorithm that has to solve the resulting MDP. It is then important to keep such an overhead to a minimum.

Definition 11.5. *Given an NMRDP M with state space S and an equivalent MDP M' with state space S', the* state overhead *of M' on M is $|S'| - |S|$.*

In this section, we propose a novel construction for the extended MDP, that achieves a significantly smaller state overhead by using reward transducers introduced in the previous section, instead of DFAs, to assign rewards. In particular, we can define an MDP that plays the same role as the one described in (Brafman, De Giacomo, and Patrizi, 2018), with the exception that it does not keep track of which formula the reward comes from. We use a Moore reward machine, which is the sum of the Moore machines for the single rewards—rather than the cross product of the DFAs for the reward formulas.

Consider an NMRDP $M = \langle S, A, Tr, \{(\varphi_i, r_i)\}_{i=1}^m \rangle$, and let $\mathcal{M}_o^i$ be the Moore reward machine for (φ_i, r_i). We define $\mathcal{M}_o = \langle Q_o, 2^{\mathcal{P}} \times A, \mathcal{R}_o, q_0, \delta_o, \theta_o \rangle$ as the Moore reward machine obtained by minimising the sum of all the other Moore reward machines, i.e., $\mathcal{M}_o$ is the minimum machine equivalent to $\mathcal{M}_o^1 + \cdots + \mathcal{M}_o^m$. We derive the new MDP $M_o = \langle S_o, A, Tr_o, R_o \rangle$ as follows:

- $S_o = S \times Q_o$;
- $Tr_o : S_o \times A \times S_o \to [0, 1]$ is defined as:

$$Tr_o((q, s), a, (q', s')) = \begin{cases} Tr(s, a, s') & \text{if } \delta_o(q, (s, a)) = q' \\ 0 & \text{otherwise;} \end{cases}$$

- $R_o : S_o \times A \to \mathbb{R}$ is defined as:

$$R_o((q, s), a) = \theta_o(\delta_o(q, (s, a)))$$

We have that M_o is equivalent to M' as for Theorem 11.1, and hence to M. We formalize this observation in the following theorem.

Theorem 11.6. *The NMRDP M and the MDP M_o are equivalent.*

Proof. We need to prove that M_o is equivalent to M', since the equivalence between M and M' is a consequence of Theorem 11.1. Notice that, by construction, S_o is isomorphic to S', and so is Tr_o to Tr', due to the definition of δ. Finally, notice that $\theta_o(q) = \sum_i \theta^i_o(q_i)$, where $\theta^i_o(q_i) = r_i$ when $q_i \in F_i$, so R_o is simply a compact representation of R'. □

The minimisation step in the construction above is important. Even assuming that the machines $\mathcal{M}^i_o$ are minimum, their sum machine may not be; thus, the minimisation step is required to minimise the state space of the resulting MDP M_o, hence the state overhead of M_o on M.

Now that we have defined the extended MDP construction based on Moore machines we show that such a construction can significantly reduce the state overhead. In fact, it can achieve an exponential improvement (in fact, factorial), as argued in the following theorem.

Theorem 11.7. *For every $n \geq 1$, there is an NMRDP M such that (i) the equivalent MDP M (as in Theorem 11.1) has state overhead $\Omega(n!)$ on M, and (ii) the equivalent MDP M_o (as introduced this section) has state overhead $\mathcal{O}(n)$ on M.*

Proof. Consider a set of propositions $\mathcal{P} = \{p_1, \ldots, p_n\}$ and an NMRDP $M = \langle 2^{\mathcal{P}}, \{\mathsf{ins}, \mathsf{del}\} \times \mathcal{P}, Tr, \{(\varphi_i, 1)\}_{i=1}^n\}$ where each φ_i is of the form:

$$\mathsf{O}^i(\neg p_1 \wedge \cdots \wedge \neg p_{i-1} \wedge p_i \wedge \neg p_{i+1} \wedge \cdots \wedge \neg p_n)$$

and Tr consists of transitions

$$(s, (\mathsf{ins}, p)) \mapsto (s \cup \{p\}) \quad \text{and} \quad (s, (\mathsf{del}, p)) \mapsto (s \setminus \{p\}).$$

for each $p \in \mathcal{P}$ and each $s \in 2^{\mathcal{P}}$. Intuitively, we can insert and delete propositions to/from states, and at the i-th step we get rewarded if p_i is true and the other propositions are false. The minimum DFA for φ_i has $\Omega(i)$ states. As a result, the MDP M' has state overhead $\Omega(n!)$. Now we argue that the state overhead of M_o is $O(n)$. A Moore reward machine $\mathcal{M}^i_o$ for $(\varphi_i, 1)$ has states $s_0, \ldots, s_i$, and a transition from s_j to s_{j+1} for each $j \leq i - 1$ and each input symbol, and it outputs 0 in all transitions but the last one, where it outputs 1 if it reads $\{p_i\}$. The minimum reward machine $\mathcal{M}_o$ equivalent to the sum of the machines $\mathcal{M}^i_o$ has states $s_0, \ldots, s_n$, and transitions from s_j to s_{j+1} for each $j \leq n - 1$ and each input symbol, with the output at the i-th transition being 1 for input $\{p_i\}$, and 0 otherwise. □

Moreover the approach based on transducers never does worse than the one based on DFAs.

Theorem 11.8. *For every NMRDP M, the equivalent MDP M_o (as introduced in this section) has state overhead smaller than or equal to the state overhead of the MDP M' (as in Theorem 11.1).*

Proof. It suffices to notice that in both cases we can build an extended state space based on the cross product of the states of automata for the reward formulas. □

Considering that our goal is to keep the state overhead to a minimum, we next focus on Mealy reward machines. Mealy machines will allow us to save on states, since they can represent Moore machines using possibly less states and never more. In particular, we define a construction that leverages Mealy reward machines, instead of Moore reward machines. Note that every Moore machine can be transformed into a Mealy machine by composing its output function with its transition function. Hence, from the MDP $M_o = \langle S_o, A, Tr_o, R_o \rangle$, we can construct a new MDP $M_e = \langle S_e, A, Tr_e, R_e \rangle$ where everything is defined as in M_o except $R_e : S_e \times A \to \mathbb{R}$ that is defined as:

$$R_e((q, s), a) = \theta_e(\delta_o(q, (s, a)))$$

Theorem 11.9. *The NMRDP M and the MDP M_e are equivalent.*

Proof. By construction, M_e is equivalent to M_o, and by Theorem 11.6 and Theorem 11.1, the thesis follows. □

11.5 Rewards as Temporal Specifications

In this section we argue for the case of the temporal logics $\text{LDL}_f/\text{LTL}_f$ as an appropriate language to specify rewards. In particular, they capture Markovian rewards without loss of efficiency (using transducers) and they can be more succinct.

Capturing Markovian rewards. We start by showing how any MDP M can be represented as an NMRDP M_{nmr} without loss of efficiency in terms of number of states. Specifically, we mean that M_{nmr} has the same states as M, and M_{nmr} can be automatically encoded back into an MDP M' which has again the same states of the original MDP M. Consider an MDP $M = \langle S, A, Tr, R \rangle$. Such an MDP is captured by the following NMRDP:

$$M_{\text{nmr}} = \langle S, A, Tr, \{(\varphi_{(s,a)}, R(s, a))\}_{s \in S, a \in A} \rangle$$

where $\varphi_{(s,a)}$ has the form $\Diamond(s \wedge a \wedge \mathit{Last})$—note that $R(s, a)$ is the Markovian reward when the last state and action are s and a, respectively. First, we argue that M_{nmr} correctly encodes the given MDP M.

Theorem 11.10. *The MDP M and the NMRDP M_{nmr} are equivalent.*

Proof. By construction, the non-Markovian rewards depend only on the last state-action pair, i.e., they are Markovian, although formalized as non-Markovian. Hence, from a non-Markovian policy $\bar{\rho}$, we can build an equivalent Markovian policy ρ by ignoring the history but the last state. Analogously, from ρ we can define a $\bar{\rho}$ such that for all the possible traces $\pi = \langle (s_0, a_1), \ldots, (s_{n-1}, a_n) \rangle$ that end up in state s_n, we have $\bar{\rho}(s_0, \ldots, s_n) = \rho(s_n)$. □

For further clarity, in Figure 11.3 is depicted the DFA corresponding to a generic $\varphi_{(s,a)}$. Then, we can convert the NMRDP M_{nmr} into an equivalent MDP M_e using a construction based on a Mealy machine, as discussed in Section 11.3. The involved

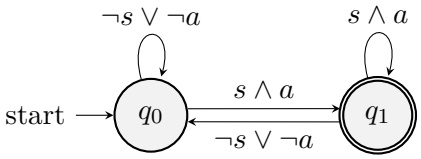


Figure 11.3. Automaton corresponding to $\varphi_{(s,a)} = \Diamond(s \wedge a \wedge last)$

Mealy machine is a straightforward state-less encoding of R, and it is depicted in Figure 11.4. Most importantly, M_e has state space $S_e = S \times Q_e = S \times \{q_0\}$ that is isomorphic to the original state space S, given the fact that Q_e is a singleton. This shows two points in favour of our transducer-based approach: (*i*) it is able to fully capture Markovian reward functions, at no cost of additional states; (*ii*) it is a significant improvement over the construction based on DFAs (Brafman, De Giacomo, and Patrizi, 2018) (see Theorem 11.1 in the background section), since it allows to handle Markovian rewards seamlessly without incurring an exponential blow-up of the state space (which in the DFA-based approach is due to the Cartesian product of the reward automata $\mathcal{A}_{\varphi_i}$).

Capturing reward transducers. Temporal specifications capture Moore (and Mealy) reward machines. To see this, consider a Moore reward machine $\mathcal{M}_o = \langle Q, \Sigma, \mathcal{R}, q_0, \delta, \theta_o \rangle$. For each $q \in Q$, let $\mathcal{A}_q$ be the DFA $\langle Q, \Sigma, q_0, \{q\}, \delta \rangle$, and let $\varphi_q = \langle \varrho_q \rangle End$ with ϱ_q a regular expression that captures the language $\mathcal{L}(\mathcal{A}_q)$. Notice that $\mathcal{M}_o$ is captured by the temporal specification $\{(\varphi_q, \theta_o(q))\}_{q \in Q}$. Mealy reward machines can be captured as well, since they can be converted into Moore reward machines. On the other hand, specifications can always be compiled into a reward transducer, as discussed in Section 11.3. Furthermore, temporal specifications can be more succinct than Moore (and Mealy) reward machines, as shown in the following theorem.

Theorem 11.11. *There is a family of non-Markovian rewards $\bar{R}_n$ that admit an* LTL$_f$ *specification of size $\mathcal{O}(n^2)$ and only reward machines of size $\Omega(2^{2^n})$.*

Proof. Consider a set A of at least two actions that an agent can perform. In addition, the agent can perform an action e(nd) that marks the end of a sequence of actions, and can observe a c(ommand). We use a construction from (Kupferman and Vardi, 2005), adapted to finite traces in (De Giacomo and Rubin, 2018). The construction consists of the regular language

$$\mathcal{L}_n = \{(A + e)^* \cdot e \cdot \mathbf{s} \cdot e \cdot (A + e)^* \cdot c \cdot \mathbf{s} \cdot e^+ \mid \mathbf{s} \in A^n\}$$

and of its LTL$_f$ specification

$$\begin{aligned} \varphi_n = \; & \left(\neg c \,\mathcal{U}\, \left(c \wedge \textstyle\bigwedge_{i=1}^{n} \bigcirc^i \left(\bigvee_{a \in A} a\right) \wedge \bigcirc^{n+1} \Box e\right)\right) \wedge \\ & \Diamond \left(e \wedge \textstyle\bigwedge_{i=1}^{n} \bigvee_{a \in A} \bigcirc^i a \wedge \Box \left(c \rightarrow \bigcirc^i a\right)\right). \end{aligned}$$

The construction has two key properties: (*i*) the size of φ_n is $\mathcal{O}(n^2)$, and (*ii*) the size of the minimum DFA for $\mathcal{L}_n$ is $\Omega(2^{2^n})$ (Chandra, Kozen, and Stockmeyer, 1981). Consider now the reward function $\bar{R}_n$ that assigns reward 1 to traces in $\mathcal{L}_n$. Intuitively, we reward the agent to initially perform sequences of actions from A with each sequence ended by the action e, and then, upon seeing the command c, perform

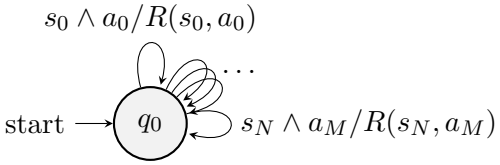


Figure 11.4. Mealy reward machine that encodes the Markovian reward function R. It has a unique state with a self-loop for each domain value of R.

a sequence $\mathbf{s}$ of length n already performed before. The reward $\bar{R}_n$ is captured by the specification $(\varphi_n, 1)$ which has size $\mathcal{O}(n^2)$, and every reward machine for $\bar{R}_n$ has size $\Omega(2^{2^n})$ since it is at least as big the minimum DFA for $\mathcal{L}_n$. □

Specifying common rewards in logic. It is often the case that we can represent the same transition-based reward function R with much less effort, by specifying a non-Markovian reward fully specified by a much more intelligible LTL$_f$ formula. For example, consider the Mountain Car environment (Moore, 1991) a well-known problem in the RL literature and in the research community. The traditional transition-based reward function R is usually implemented using the If-This-Then-That pattern (IFTTT), namely *"as long as the car has not reached the goal, give reward -1.0"*. Such statement in natural language can be formalized into a temporal specification, whilst being relatively close in terms of readability to its original formulation. By Theorem 11.10, we know we can always represent such reward function with temporal specifications. The reward function of that environment can be represented by the specification $(\varphi, -1.0)$ where $\varphi = \neg\Diamond p$ and p means *the car has reached the goal*, a proposition opportunely extracted from the state space. By translating the formula into a DFA $\mathcal{A}_\varphi$ and employing the compilation into the equivalent MDP as explained in Section 11.2, we make such non-Markovian reward learnable by classic RL algorithms. Thus, we can specify rewards using high-level formal specifications, and then compile them automatically into standard models compatible with solvers.

To pursue the analogy with software engineering: the "raw" R is binary language, the equivalent $\{(\varphi_{(s,a)}, R(s,a))\}_{s \in S, a \in A}$ is the *decompiled* program in a high-level language, say the C++ language, and the LTL$_f$ formulas are programs written in that language. We advocate that temporal specifications using proper formal languages becomes the standard for reward engineering.

11.6 Monitoring Rewards

In this section, we define monitoring rewards, an extension of temporal rewards based on the four satisfaction conditions from runtime monitoring on finite traces (De Giacomo, Masellis, Grasso, et al., 2014). They provide an additional layer of abstraction which allows one to focus on one condition φ and to assign different rewards based on how the current trace satisfies φ. Furthermore, in some cases, monitoring rewards allow one to derive the value of future rewards, giving additional guidance to the learning process.

Example 11.12. *The condition "never p or eventually q" can be temporarily true, temporarily false, or permanently true. We can define a monitoring reward that*

returns 1 *when the condition is temporarily true,* -1 *when temporarily false, and* 10 *when permanently true.*

Definition 11.13. *A* monitoring reward *is a 5-tuple* $\langle \varphi, r, c, s, f \rangle$ *where* φ *is a temporal formula and* r*,* c*,* s*,* f *are integers; we call* φ *the* reward formula *and* r, c, s, f *the* reward values*.*

When a monitoring reward of the form above is specified, an agent receives a reward value r (reward) when φ is temporarily true in the current partial trace, c (cost) when it is temporarily false, s (success) when permanently true, and f (failure) when permanently false. We call each of the former cases a *reward condition*. If not stated otherwise, we assume that $r \geq 0$, $s \geq 0$, $c \leq 0$ and $f \leq 0$, as we consider this the natural interpretation of the four conditions. If multiple monitoring rewards are given at the same time, then the agent receives the sum of the values computed for each monitoring reward. We can now formalise the monitoring reward given in the previous example.

Example 11.14. *Consider the monitoring reward:*

$$\langle (\Box \neg p) \vee (\Diamond q), 1, -1, 10, 0 \rangle.$$

If p *does not hold anytime in the current trace, and the same holds for* q*, then the agent receives reward* 1*. If* p *does hold sometimes in the current trace, and* q *does not hold anytime, then the agent receives* -1*. If* q *holds sometimes in the current trace, then the agent receives* 10*.*

We have that exactly one of the reward conditions is true at any moment, because a formula is either temporarily true, temporarily false, permanently true, or permanently false in a trace.

Theorem 11.15. $\pi \models [\![\varphi = \mathcal{T}]\!]$ *holds for exactly one* $\mathcal{T}$ *from* $\{temp\_true, temp\_false, perm\_true, perm\_f$

Proof. We have that $\pi \models \varphi$ implies that either $\pi \models [\![\varphi = temp\_true]\!]$ or $\pi \models [\![\varphi = perm\_true]\!]$, and similarly for the dual case $\pi \not\models \varphi$. Then, the theorem follows from the fact that either $\pi \models \varphi$ or $\pi \not\models \varphi$. □

When the reward condition is permanently true (or false) in the current trace, the agent will keep receiving the same reward value. In fact, the reward condition will be permanently true (resp., false) at any future step, and in particular will not become temporarily true or false.

Theorem 11.16. *For* $\mathcal{T} \in \{perm\_true, perm\_false\}$*, we have that* $\pi \models [\![\varphi = \mathcal{T}]\!]$ *implies* $\pi\pi' \models [\![\varphi = \mathcal{T}]\!]$ *for every trace extension* π'*—and hence* $\pi\pi' \not\models [\![\varphi = \mathcal{T}']\!]$ *for* $\mathcal{T}' \in \{temp\_true, temp\_false\}$*.*

As a consequence of the previous theorem, if we are interested in traces of a fixed length (e.g., episodes in RL), the total reward value on a trace can be computed as soon as the reward condition becomes permanently true or false. This ability can be used to better guide the learning process.

Each monitoring reward (φ, r, c, s, f) admits the equivalent *dual form* $(\neg\varphi, c, r, f, s)$, where we negate the formula and swap values for reward and cost, and for success and failure. To see the equivalence, it suffices to note that a φ is temporarily/permanently true iff $\neg\varphi$ is temporarily/permanently false.

Theorem 11.17. *Monitoring rewards* (φ, r, c, s, f) *and* $(\neg\varphi, c, r, f, s)$ *return the same value on every trace.*

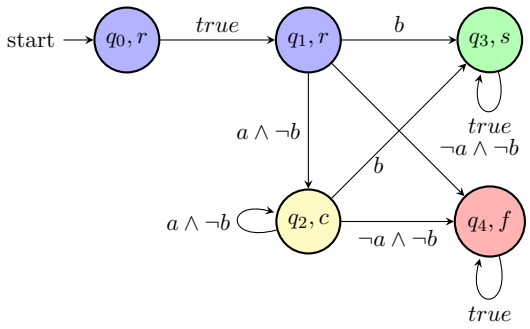


Figure 11.5. Moore machine for $\langle \bullet(a \,\mathcal{U}\, b), r, c, s, f \rangle$. States are color-coded based on the condition they monitor: purple for *temp_true*, yellow for *temp_false*, green for *perm_true*, and red for *perm_false*.

Monitoring rewards capture $\text{LDL}_f/\text{LTL}_f$ specifications as in (Brafman, De Giacomo, and Patrizi, 2018). Specifically, a specification (φ, r) can be restated as the monitoring reward $(\varphi, r, 0, r, 0)$. What is less obvious is that each monitoring reward can be expressed as a set of four LDL_f specifications. In fact, the four reward conditions can be directly expressed in LDL_f without any meta-logical machinery (De Giacomo, Masellis, Grasso, et al., 2014).[2] So a reward (φ, r, c, s, f) can be restated as a set of specifications $\{(\varphi^r, r), (\varphi^c, c)(\varphi^s, s), (\varphi^f, f)\}$.

Since LDL_f rewards capture monitoring rewards, we can turn an NMRDP with monitoring rewards into an equivalent MDP using the extended MDP construction based on DFAs (Brafman, De Giacomo, and Patrizi, 2018). We argue that this construction introduces an unnecessary state overhead in the case of monitoring rewards. In fact, the formulas $\varphi^r, \varphi^c, \varphi^s, \varphi^f$ (and also φ) admit shape-equivalent DFAs, as an immediate consequence of Theorem 3 and Corollary 1 of (De Giacomo, De Masellis, et al., 2020). Thus, the corresponding Moore reward machines can be combined into a single machine by taking the direct sum—see Section 11.3. In particular, the resulting reward machine has the same number of states as the DFA for φ, and hence we have the same state overhead of a simple reward specification (φ, r_φ).

Example 11.18. *Consider the monitoring reward:*

$$\langle \bullet(a \,\mathcal{U}\, b), r, c, s, f \rangle.$$

In Figure 11.5 we show the equivalent Moore reward machine. This is the result of a direct sum between 4 Moore machines, where each of them models one condition at a time. The conditions are highlighted with different colors per state.

As a result, monitoring rewards introduce no additional state overhead compared to simple temporal rewards.

Theorem 11.19. *If an NMRDP $\langle S, A, Tr, (\varphi, r_\varphi) \rangle$ admits an equivalent MDP which introduces a state overhead n, then every NMRDP of the form $\langle S, A, Tr, (\varphi, r, c, s, f) \rangle$ admits an equivalent MDP which introduces state overhead n.*

[2] Note that we need LDL_f and not simply LTL_f because we need to represent prefixes of traces.

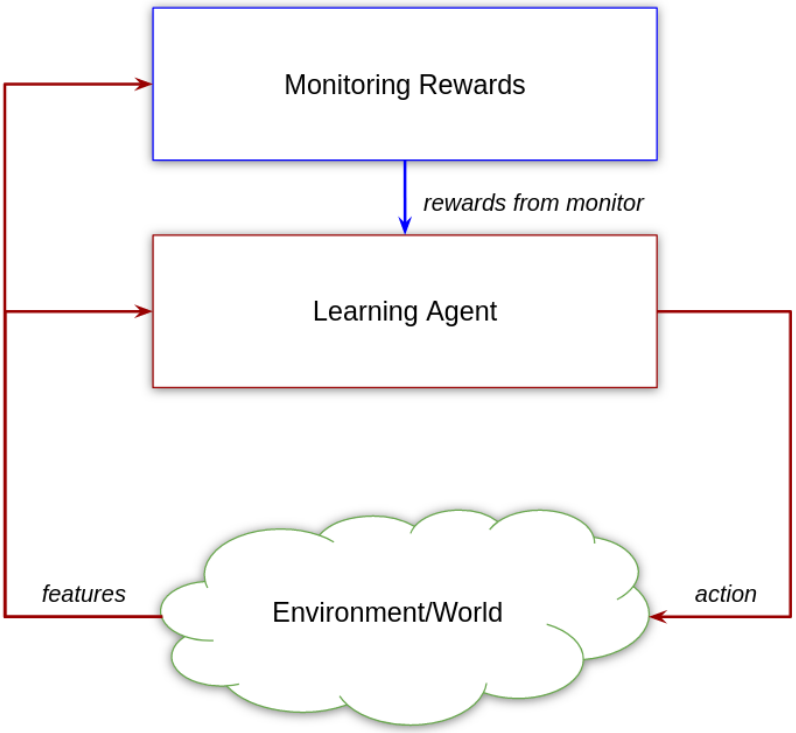


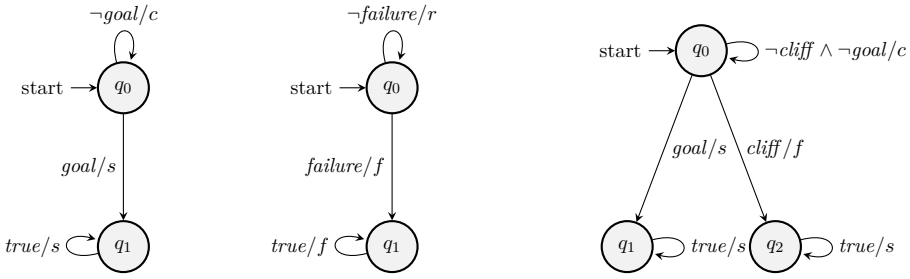
Figure 11.6. An RL scenario with monitoring rewards.

11.7 Applications in RL

One field that can benefit from the approach described is Reinforcement Learning. Indeed, most RL algorithms assume the underlying hidden model to be an MDP. Hence, the approach described in the previous sections can be very useful for RL. The idea is that we can give a specification at a high level of abstraction on how to give rewards. The rewards are then given by the induced reward transducer, as explained earlier in this work. Moreover, being the overhead the smallest possible, algorithms on such MDPs are more effective.

Reward engineering is a very crucial task when devising RL domains. The specification of reward functions can be cumbersome and error-prone, breaking RL algorithms in surprising, counterintuitive ways. This phenomenon is known in the community as *reward hacking* (Amodei et al., 2016). An illustrative example is shown in (Clark and Amodei, 2016). Here, we propose that the experiment designer can specify monitoring of temporal specification to have a finer control on the reward given to the agent, despite having a concise, human-friendly language like $\text{LTL}_f/\text{LDL}_f$. We argue that is more convenient to think in terms of monitoring rewards of the form (φ, r, c, s, f) than temporal specifications of the form (φ, r).

In Figure 11.6 is depicted the scenario we have in mind. The world states are described by propositional features. The agent acts in an environment and observes such features to take the next action. Each observation is passed to a monitor that interpret the observation, updates its state and produces a reward signal that is then given to the agent. In this way, the agent's behaviour is implicitly driven by the monitor via rewards, specified at high-level by the designer. In the rest of this



(a) $\Diamond goal$ **(b)** $\Box \neg failure$ **(c)** $\neg cliff \, \mathcal{U} \, goal$

Figure 11.7. In (a) the Mealy reward machine for $(\Diamond goal, r, c, s, f)$, in (b) the Mealy reward machine for $(\Box \neg failure, r, c, s, f)$, and in (c) the Mealy reward machine for $(\neg cliff \, \mathcal{U} \, goal, r, c, s, f)$.

section, we will describe potential applications of our approach.

Mountain Car. The Mountain Car environment (Moore, 1991) is a classic RL problem. The state space is the set of pairs $\langle position, velocity \rangle$. A reward of -1 is given at each timestep. The goal state is when $position \geq 0.5$. We model the reward function with a monitoring temporal specification $(\Diamond goal, 0, -1, 0, 0)$, where *goal* is a fluent that is true when $(position \geq 0.5)$, false otherwise. The training is performed over the extended MDP, where the state space is the cross product between the original MDP state space and the Mealy reward machine, shown in Figure 11.7a. Notice that the reward assignment is completely handled by the framework, according to the current simulation of the machine, in a given episode. Specifications of the form $\Diamond p$, when p is a state formula, are useful to capture *achievement* goals, i.e. a condition that must be satisfied in the future, before the end of the trace.

Cart Pole. In the Cart Pole environment (Barto, Sutton, and Anderson, 1983), the goal is to prevent a pendulum from falling over. The state space is the set of tuples $\langle position, velocity, pole\_angle, pole\_velocity \rangle$. A reward signal of $+1$ is given at each time step. The failure states are the ones satisfying $|pole\_angle| \geq 12°$ or $|position| \geq 2.4$. We model the reward function with a monitoring temporal specification $(\Box \neg failure, +1, 0, 0, 0)$, where *failure* is true in failure states. The associated Mealy reward machine is shown in Figure 11.7b. Specifications of the form $\Box q$, when q is a state formula, are useful to capture *maintenance* goals, i.e., conditions that must be satisfied until the end of the trace.

Cliff Walking. A task that is both achievement and maintenance is the *approach-avoid* task, expressed by the formula $\neg failure \, \mathcal{U} \, goal$. An instance of such task is the Cliff Walking environment in Ch. 6 of (Sutton and Barto, 1998). In this kind of gridworld, the reward is -1 on all transitions except those into a special region at the bottom of the grid, representing "the cliff". Stepping into this region incurs a reward of -100 and makes the simulation to fail. The goal is to reach a specific goal state, whereas the cliff region constitutes the set of failure states. Such reward function can be captured by the monitoring specification $(\neg cliff \, \mathcal{U} \, goal, 0, -1, +1, -100)$. Other examples of environments that can be modeled in the same way are Frozen Lake (OpenAI, 2016), the 4x3 world in Ch. 21 of (Russell and Norvig, 2010), and WaterWorld domain (Karpathy, 2015).

Taxi domain. In the Taxi domain (Dietterich, 2000) there are 4 locations and the goal is to pick up the passenger at one location (the taxi itself is a possible location) and drop him off in another. The reward is +20 points for a successful drop-off, and -1 point for every timestep it takes. There is also a -10 reward signal for illegal pick-up and drop-off actions. The goal state is when the passenger is dropped off at the right place. We can model the Taxi problem as a sequence task: $(\Diamond(p \wedge \Diamond q), 0, -1, +20, 0)$, where p means "pick up the passenger" and q means "drop-off the passenger to the right location". The bad action penalty is another temporal specification $(\Diamond bad\_action, -10, 0, 0, 0)$. Although we use two temporal specification, we remind that both get compiled into a more compact single Mealy reward machine. Other RL environments that have sequential tasks are the Minecraft environment (**andreas2017modular**), the task to break columns in order in Breakout or to visit colors in Sapientino (De Giacomo, Iocchi, et al., 2019).

11.8 Summary and Discussion

In this work we have formalised the notion of *overhead* as the state extension introduced to describe an NMRDP in the form of an MDP. We have considered the overhead introduced by approaches that directly use the DFAs for the reward formulas (Brafman, De Giacomo, and Patrizi, 2018), and argued that part of that overhead is unnecessary if we are not interested to know which reward specifications are accountable for the rewards assigned at any given moment. We have shown that, giving up that information, approaches based on reward machines can build exponentially (in fact factorially) smaller extended MDPs, while never doing worse than direct use of DFAs. We have argued that the temporal logics $\text{LDL}_f/\text{LTL}_f$ are an appropriate language to specify rewards, and then extended temporal specifications to *monitoring specifications*, which build on the four classic monitoring conditions allowing a reward designer to assign rewards based on temporary/permanent satisfaction of a temporal formula. We have shown how transducer-based approaches allow for implementing monitoring specifications at no extra cost compared to reward specifications; in other words, the extension from one condition to four conditions comes for free. Finally, we have applied monitoring rewards to reinforcement learning, showing how they can be used to capture the reward functions of popular RL environments. As a future work, we would like to provide a reasoning service that reports to the user whether one of the monitoring condition cannot be satisfied.

Chapter 12

Domain-independent reward machines for modular integration of planning and learning

Integrating planning and learning components has many advantages in practical applications, as it allows for combining the different benefits of the two approaches: prediction of future states from planning with adaptivity to current situations from learning. However, a problem with is approach is that the two components should share a common representation of the information about the environment (e.g., states and actions). Previous work addresses this problem in the case where planning and learning are defined over different state variables, by defining a joint state space and a mapping between the two representations. In this chapter, we present a method for integrating planning and reinforcement learning using a modular design where the two components can use their own representation formalism, without requiring an explicit mapping between them. More specifically, we introduce the concept of domain-independent reward machines, generated by a goal-oriented planning system and use them to drive a reinforcement learning agent to reach a goal state. Moreover, we show how to automatically generate and use sub task decomposition to speed up the reinforcement learning process.

The rest of this chapter is structured as follows:

- Section 12.1 introduces the problem and gives the motivations for the work.
- Section 12.2 briefly surveys relevant and similar work.
- Section 12.3 formalizes the problem,
- Section 12.4 provides a solution to the problem.
- Section 12.5 shows the advantages of the approach in several use cases.
- Section 12.6 concludes the chapter.

The contents of this chapter have been published in the workshop paper (De Giacomo, Favorito, Iocchi, et al., 2021).

12.1 Introduction

Reinforcement Learning (RL) (Sutton and Barto, 1998) is a powerful tool for computing optimal behaviors of an agent, by collecting experience during the execution of some task and without requiring any knowledge about the environment's model. Many algorithms have been developed to explore the environment in an efficient way. Some model-based approaches aim at reconstructing the underlying dynamic system, typically a Markov Decision Process (MDP), during the learning phase. A more recent one also aims at extracting knowledge (using a symbolic formalism) from RL trials.

Hierarchical RL (HRL) is an extension of RL, where the problem is organized in a hierarchy of sub-problems, with the aim of speeding up the learning process. The use of factored MDPs, i.e., where states are modeled using state variables, in model-based RL allows for introducing a-priori knowledge expressed in a symbolic formalism.

In these approaches, the model is fully specified, and this includes mappings that relate the various hierarchical levels. However, HRL typically takes advantage of Knowledge Engineering (i.e., knowledge representation and reasoning tools) to speedup the learning process, rather than for specifying goals.

Very recent works have adopted logical specifications to express (temporal) goals. When temporal logics, such as LTL , are used, problems involving Non-Markovian rewards can easily be formalized (i.e., Non-Markovian Reward Decision Processes, NMRDP). Examples of such approaches include Reward Machines (Icarte, Klassen, et al., 2018b) and Restraining Bolts (De Giacomo, Iocchi, et al., 2019), which exploit finite-state machines as a way to specify RL agents' rewards.

When hierarchical structures are considered, several mechanisms to speed up the learning process can be used, such as options (Sutton, Precup, and Singh, 1999), policy sketches (Andreas, Klein, and Levine, 2017), etc. However, such previous approaches require additional modelling effort and, in hierarchical settings, also a mapping between the different representation layers. Automatic generation of sub tasks in HRL is still an open problem.

In this work, we start from the work described in (Icarte, Klassen, et al., 2018b), where HRL, non-Markovian rewards, and task decomposition, are combined into the Reward Machine framework. Similarly to that, we also consider two separate representation layers, one for the goal and one for the learning agent, but do not require any mapping between them. Specifically, we consider the setting depicted in Fig. 12.1, where three distinct modules are used to learn an optimal behavior (policy) over the environment:

- a *planning module*, which performs high-level *offline* reasoning and generates a plan π, used to guide, and speedup, the learning process;
- a *reward machine*, which is, essentially, a runtime monitor that observes the environment from the high-level perspective and returns a signal σ whenever the input plan π advances one step towards the goal;
- a *learning module*, which observes the environment from a low-level perspective, interacting with it through actions and possible rewards, and receiving the signals σ from the reward machine.

The two perspectives of the environment are based on the use of two independent sets of sensors, S_1 and S_2, each extracting its own state variables (or features) of the environment.

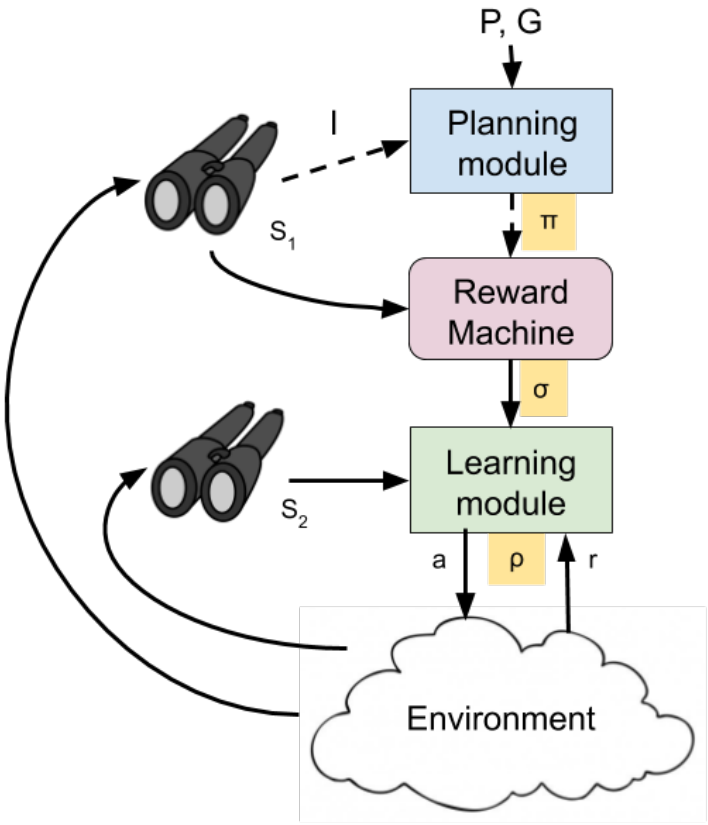


Figure 12.1. Architectural schema

Notice that the planning module operates offline and state variables observed by S_1 are used to detect the initial state I (dashed line in Fig. 12.1) to start the plan generation process. At plan execution time, the state variables observed by the reward machine through S_1 take values consistently with the observations of the learning agent through S_2.

As said, this setting is very similar to that of (Icarte, Klassen, et al., 2018b), with the crucial difference that we do not require an explicit mapping between the two representation layers and thus the reward machine does not need to be an input for the RL algorithm but can be kept as a separate component. In other words, in our work the RL agent does not know the reward machine, but it just receives domain-independent signals from it. Of course, an implicit mapping exists, as the two representations derive from observations of the same environment, however it is not necessary to make this mapping explicit to the RL agent and thus the proposed approach does not require additional modeling efforts. Nonetheless, in some cases, in order to ensure such implicit relation, we may require the set of sensors to be synchronized (i.e., the components should share a common clock).

Since the learning component does not require a representation of the domain model specified at the planning level and, consequently, the reward machine can produce only signals not depending on the state representations used by the planning and learning modules, both components are referred to as *domain-independent*. Such *domain-independent* components can thus be designed and implemented separately, allowing the system to be highly modular. For example, planner modules with

different representations of states and actions can be interchanged without requiring any modification of the implementation of the learning module; the same reward machine can be applied to different learning agents with different representations of states and actions without requiring any modification on the agent.

In this work, we show that, although the planning and learning modules are loosely linked with domain-independent signals only, they can still cooperate to reach a common goal, as they observe and act (directly or indirectly) on the same environment. We also discuss how such mechanism allows for an easy way to automatically define and exploit sub task decomposition, to speedup the learning process and, finally, report on experimental results. The original contributions of this work are the following.

Firstly, we propose an approach for the integration of a planning component (or more in general a reasoning system) with a RL agent, in a setting where the components use different representation formalisms (for example, different state variables and different actions), without requiring an explicit mapping. This simplifies the previous approach based on reward machines (Icarte, Klassen, et al., 2018b) as reducing the required modeling effort and, more importantly, broadens the range of applicability of the approach, to those situations where a mapping is not simple to define or not possible at all.

Secondly, we present a mechanism to automatically generate sub task decomposition, that can be used to speedup the learning process, by extending previous work on restraining bolts (De Giacomo, Iocchi, et al., 2019), demonstrating faster convergence when sub task decompositions are considered.

12.2 Related work

A general approach for integrating planning and learning is given by model-based RL where the goal is to reconstruct the model of the environment while learning. Dyna (Sutton, 1990) and R-Max (Brafman and Tennenholtz, 2002) are example of such methods in which the learning experience is used to build a model of the environment and such a model is used to generate policies followed during the learning process. The use of inaccurate models and few real trials to speed-up learning is also presented in (Abbeel, Quigley, and Ng, 2006). In all these works the behavior of the agent is guided by a reward function that is assumed to be available and sampled during agent execution.

The use of a planner to drive the RL process is shown for example in (Grzes and Kudenko, 2008; Efthymiadis and Kudenko, 2014) where it is used to define a reward shaping functions to drive the agent along the plan, and in (Leonetti, Iocchi, and Stone, 2016) to constrain the exploration space of the agent, by defining partial policies in which each state is associated with a set of possible actions that will be considered during the exploration phase of RL. In these works, the planning domain is defined on the same state representation that is used by the RL algorithm and it is thus deeply linked to it.

Hierarchical Reinforcement Learning (HRL) and options (Sutton, Precup, and Singh, 1999) are also commonly used to speed-up the learning process. While, in general, the models used at the layers of the hierarchical architecture and the options are manually defined, there have been some approaches to generate them automatically with a planning component, such as (Grounds and Kudenko, 2007; Yang et al., 2018). Also in these cases, either the planning and the learning components share the same representation of the states, or an explicit mapping between these representations is required.

Finally, more general approaches to drive the RL process of an agent, considering also temporal goals and non-Markovian rewards are reward machines (Icarte, Klassen, et al., 2018b) and restraining bolts (De Giacomo, Iocchi, et al., 2019). The use of an automated planner to generate controllers for reward machines is also presented in (Leon Illanes et al., 2019), but again an explicit mapping between the representations used by the planner component and the learning agent is required. While the restraining bolts described in (De Giacomo, Iocchi, et al., 2019) use a different representation with respect to the one used by the agent, without requiring an explicit mapping between such representations. However, this work does not describe the generation of the bolt that is assumed to be given.

The method described in this work combines the advantages of previous works by defining a framework for automatic generation of a reward machine using model-based and goal-oriented planning, in order to drive the RL agent to learn a policy following the desired plan, thus achieving the desired goal. We solve this problem in the setting in which the planning component and the learning component use different representations of states and actions involved in the task and a mapping between such representation is not required. Such a reward machine, whose states must not be mapped to the states of the learning agent, is called in this work *domain-independent* to emphasize its modularity. Indeed, as shown later, a *domain-independent reward machine* communicates with the RL agent through domain-independent messages (i.e., messages not expressed in terms of the representation used to formalize the planning problem). Consequently, a domain-independent reward machine can be placed on any RL agent without requiring additional specifications or modifications of such an agent.

12.3 Problem formulation

The hierarchical architecture we consider is reported in Fig. 12.1. A planning module generates a plan π, given a domain model and a goal G. The plan is then used to generate a *reward machine* (see below), to guide the learning process of a learning agent, by providing suitable signals σ during the learning process. We next introduce the basic components.

Planning module. A *(deterministic) planning domain* is a tuple $\mathcal{P} = \langle V, A_D \rangle$, with V a finite set of boolean state variables and A_D a set of action descriptions (e.g., in terms of preconditions and effects over V). A state W of $\mathcal{P}$ is a (total) assignment to the variables in V, represented as a set $W \in 2^V$, s.t. $W \in V$ is true iff $W \in q$. Actions are intended to be executed in a state W and lead to exactly one (in the deterministic setting) successor state W'. Given a planning domain $\mathcal{P}$, an initial state I and a goal G (expressed as a formula over V), an automatic planner generates, if any, a *plan* $\pi = \alpha_0, \alpha_1, \ldots, \alpha_n$, i.e., a finite sequence of actions that, when executed from the initial state I, takes $\mathcal{P}$ to a state satisfying G. In our setting, the values stored in variables from V are consistent with the values returned by the sensors in S_1, which observe the environment. In this work we consider offline planning. However, the proposed approach is not limited to this case and can be adapted to online planning and replanning. In this case, the RL agent will adapt to new plans, and thus new reward machines, through experience (this requires that changes in plans occur much less frequently than RL agent's action executions).

Reinforcement learning module. A *Markov Decision Process* (MDP) is a tuple $\mathcal{M} = \langle S, A, Tr, R \rangle$ containing: a set S of states; a set A of actions; a transition

function $Tr : P(s'|s,a)$ returning, for every state s and action a, a probability distribution over the next state s'; and a reward function $R : S \times A \times S \to \Re$ that specifies the reward (a real value) received by the agent when transitioning from state s to state s' by applying action a. The states S of the MDP are observed with a set of sensors S_2 that are in general different from those in S_1, used at the planning level. A solution to an MDP is a function $\rho : S \mapsto A$, called *policy*, assigning an action to each state. The *value* of a policy ρ at state s, denoted $v^\rho(s)$, is the expected sum of the rewards obtained when starting at state s and selecting actions based on ρ (possibly discounted by a factor γ, with $0 \leq \gamma \leq 1$). Reinforcement learning agents are designed to find optimal polices over MDPs.

Reward machine. In this work, planning and learning are integrated through the definition of a *reward machine* (Icarte, Klassen, et al., 2018b) over the state variables V. This machine acts as a runtime monitor observing the running environment from the same (high-level) perspective as the planning module, and sending suitable signals σ to the learning module, depending on the advancement of the environment state wrt the plan π returned by the planning module. Formally, a *reward machine* is a tuple $\mathcal{RM} = \langle Q, q_0, \delta_q, \delta_r \rangle$, where: Q is a finite set of states; $q_0 \in Q$ is an initial state; $\delta_q : Q \times 2^V \mapsto Q$ is a state-transition function; and $\delta_r : Q \times Q \mapsto \Re$ is a reward-transition function. The reward machine is automatically generated from a plan π, as described in (Leon Illanes et al., 2019). However, in contrast with that work, we do not require: i) to define a (new) joint space state including both V from the planning module and S from the learning module, or ii) to explicitly relate S and V. Consequently, our $\mathcal{RM}$ is defined by only considering elements at the planning level (i.e., only using the state variables V captured by sensors S_1), without relating them to states S and actions A used by the learning agent. More specifically, $Q_\pi \subseteq 2^V$ contains all the states traversed during the execution of plan π, $q_0 = I$ is the initial state, δ_q is defined over transitions of state variables in V and δ_r is associated only to transitions in Q.

The problem In this work we address the problem of learning an optimal policy over the MDP $\mathcal{M}$, using the architecture described above, in particular without requiring any explicit mapping to connect the various layers. The proposed solution defines information σ to be shared between these two modules, that is domain-independent, i.e., not based on V or S.

12.4 Solution

The solution is based on the creation of a reward machine that controls the learning process of the RL agent by using only domain-independent signals. Below, we describe the steps to generate the machine and the use of options associated with it.

12.4.1 Reward machine generation

The reward machine is automatically obtained by first deriving a transition graph from the plan π generated by the planner and then by associating reward values with state transitions. Specifically, a plan π can be transformed into a transition graph $\mathcal{T}_\pi = \langle Q_\pi, q_{0_\pi}, E_\pi \rangle$ where: $Q_\pi \subseteq 2^V$ is the set of states traversed during the execution of π over the planning domain $\mathcal{P}$; $q_{0_\pi} = I \in Q_\pi$ is the initial state of the planning problem; and $E_\pi \subseteq Q_\pi \times Q_\pi$ is the set of edges (i.e., the actions occurring

in π) connecting two states in Q_π, according to the execution of π. By exploiting the reasoning capabilities of the planning system, it is possible to associate each state $q \in Q_\pi$ with a formula $\phi(q)$ over state variables V denoting the set of states that can be reached after the execution of the plan up to that state. Since we focus on classical planners generating sequential plans, the transition graph $\mathcal{T}_\pi$ is a linear graph with initial node I, one edge for each action α_i, and a final node where the goal G is satisfied.

A reward machine $\mathcal{RM}_\pi$ can now be derived from the transition graph $\mathcal{T}_\pi$ by just adding a mechanism to associate rewards with state transitions. A straightforward implementation consists in assigning a high positive rewards to the transitions reaching a goal state and zero to the other transitions. Thus, $\mathcal{RM}_\pi = \langle Q_\pi, q_{0_\pi}, \delta_q, \delta_r \rangle$, where $\delta_q(q, \phi(q')) = q'$ iff $(q, q') \in E_\pi$ and $\phi(q')$ denotes the set of states denoted by the formula $\phi(q')$ (i.e., states reached after the execution of the plan up to state q'), and $\delta_r(q, q') > 0$ if $q' \in G$ is a goal state, 0 otherwise. In practice, forms of reward shaping applied to the reward machine can help in speeding up the learning process (Camacho, Icarte, et al., 2019).

12.4.2 Use of the reward machine for RL

The reward machine continuously monitors the evolution of the plan and reports to the underlying RL agent the information necessary to guarantee that the RL agent would converge to a policy that will reach a goal state. To this end, the $\mathcal{RM}$ checks occurrence of a transition $\delta_q(q_t, \phi(q_{t+1})) = q_{t+1}$ in the current state q_t. When $\phi(q_{t+1})$ becomes true (as observed through sensors S_1), then a state transition is detected and communicated to the RL agent. Upon detecting a state-transition, the $\mathcal{RM}$ performs the following operations: 1) updates current and past states: $q_{t-1} \leftarrow q_t$, $q_t \leftarrow q_{t+1}$, 2) sends signal $\sigma_t = \langle \hat{q}_t, r_t \rangle$ to the RL agent, where: $\hat{q}_t$ is an encoding of the current machine's state q_t, and $r_t = \delta_r(q_{t-1}, q_t)$ is the reward value associated with the current machine's transition.

Importantly, observe that the encoding $\hat{q}_t$ can be any, as long as not expressed in terms of V, but in a domain-independent way. For example, it can be an integer corresponding to the index of q_t in some enumeration of Q_π.

In order to accept such information, the RL agent must be extended with a single variable to represent the encoding of the state of the reward machine (for example, an integer variable) and must take into account additional rewards coming from the reward machine. Therefore, the RL agent will act on a new MDP $\mathcal{M}' = \langle S \times \hat{Q}, A, Tr', R' \rangle$, where $S \times \hat{Q}$ is the extended space state including the encoding $\hat{Q}$ of the state of the reward machine (e.g., an integer value), Tr' and R' are the extended transition and reward functions that are unknown to the agent and thus we do not need to specify them. Notice that R' is extended by summing rewards r_t coming from the reward machine, in addition to the rewards coming from the environment. In order to guarantee reaching plan goals, we require rewards coming from the reward machine to be (significantly) higher than the rewards coming from the environment. When achieving the planning goal is the only objective of the agent, we can set to zero all the rewards coming from the environment.

We observe that the notion of $\mathcal{RM}$s is essentially analogous to that of Restraining Bolts (RB) proposed in (De Giacomo, Iocchi, et al., 2019), i.e., runtime monitors offering rewards when favorable state transitions occur. In fact, the whole setting we consider here is analogous to that of (De Giacomo, Iocchi, et al., 2019), thus we can take advantage of the results reported there. In particular, Th. 6 states that if the RL agent can accept rewards from the RB and can keep track of the RB current state, then any RL algorithm is successful in making the agent learn an optimal

policy that enforces the RB (i.e., that achieves a goal state, in our case). Since, as it can be easily seen, the MDP $\mathcal{M}'$ defind above captures exactly this situation (the reward includes the $\mathcal{RM}$'s and the agent state is extended to accommodate a representation of the current $\mathcal{RM}$ state), it turns out that we can learn an optimal policy by operating on $\mathcal{M}'$.

12.4.3 Automatic sub task decomposition

In this work we follow the sub-task decomposition induced by the reward machine, as proposed in (Icarte, Klassen, et al., 2018b), by associating different q functions to the states of the $\mathcal{RM}$. However, as a difference with QRM algorithm proposed in (Icarte, Klassen, et al., 2018b), in this work we focus on single task scenarios and we use an on-policy method. Possible use of off-policy methods to learn in parallel multiple reward machines associated to different tasks is left as future work.

Moreover, in our framework, we exploit domain-independent specifications of the $\mathcal{RM}$ and of the RL agent to implement a mechanism to enable/disable exploration for each sub-task, in order to speed-up convergence. This mechanism is similar to *options* (Sutton, Precup, and Singh, 1999) or other techniques for learning sub-tasks in hierarchical RL (Hengst, 2010)

In our implementation, sub-tasks are defined as pairs of transition (q_{t-1}, q_t) of the $\mathcal{RM}$ and the RL agent can detect start and end of a sub-task from the signals σ emitted by the $\mathcal{RM}$. Therefore, each signal received by the RL agent from the $\mathcal{RM}$ indicates the end of the previous sub-task and the start of a new one. At this moment, the RL agent can decide to enable/disable exploration for this sub-task until the next signal. When exploration is disabled, the agent actually exploits the current policy to achieve the current sub task, while when exploration is enabled, the agent learns how to improve its policy for the current sub task. This mechanism allows for speeding up the learning process by avoiding exploration steps for sub tasks for which the current policy is good enough (or optimal).

We can define different criteria for deciding when to enable/disable exploration for sub tasks, ranging from ϵ-greedy to more informed probabilistic selections. Possible criteria include: A) a constant ϵ-greedy approach, B) a variable ϵ-greedy approach considering the number of visits of the transition (q_{t-1}, q_t), C) a probabilistic choice based on percentage of success in the transition (q_{t-1}, q_t). In this work, we focus on evaluation of criterion C, since we consider environments with failure states in the reward machine that prevent the agent to proceed towards the goal and thus make the percentage of success in a transition a relevant choice.

The full procedure for extending a RL algorithm to control sub task exploration is described through the following snippets of algorithms. Here we refer to an on-policy method where the decision of enabling/disabling exploration for a sub task is applied to the policy being learned.

We make use of a variable *exploration_ON* that denotes whether exploration (i.e., choosing actions not only according to the best values of the current policy) is enabled for the current task or not. This choice is kept for the entire execution of the current sub task. On receiving a message from the reward machine, the agent chooses how to operate for the next sub task.

Variable *exploration_ON* // exploration is enabled

Function *choose_action*():

if *exploration_ON* then

    ϵ-greedy choice

else

choose_best_action

Function $on\_receive(\sigma_t)$:

$exploration\_ON$ = subtask_expl_criterion()

Function subtask_expl_criterion():

$p = Success(q_{t-1}, q_t)/Visits(q_{t-1}, q_t)$

return $random\_value(0, 1) > p$

12.5 Experimental results

To show the effectiveness of the proposed method we performed some experiments in three settings already used in previous works in RL: Breakout, Sapientino and Minecraft. These experimental scenarios have been used in (De Giacomo, Iocchi, et al., 2019) to evaluate restraining bolts, while here we consider a more general setting in which reward machines are generated by automated planning procedures. The tasks to be learned (see (De Giacomo, Iocchi, et al., 2019) for details) require the agents to perform sequences of actions. The problems can be easily modelled with a planning language and corresponding plans can be generated by suitable planners. More specifically, in Breakout we consider the goal of breaking the columns of bricks in a given order (e.g., from left to right), so the high-level plan is a sequence of actions *break_column_i*. In Sapientino, a particular sequence of actions (bip) must be executed according to the colors of the cells in which the little robot is, so the plan is a proper sequence of *goto_xy* and *bip* actions. Finally, the goal of Minecraft is to achieve 10 sub-goals by combining proper sequences of actions that must be executed in the environment at proper locations.

Notice that in these examples, state variables observed at the different layers (reward machine and RL agent) are disjoint (more details) below, but we do not need to modify the state representations of the RL agent according to the specific reward machine. In order to use existing algorithms (e.g., QRM (Icarte, Klassen, et al., 2018b)) in these domains, additional modelling and modifications are necessary: 1) extend the state representation of the RL agent by considering the state variables used in the reward machine, 2) provide the labelling function bewteen states of the RL agent and state variables of the $\mathcal{RM}$.

More specifically, in Breakout the state representation contains the paddle position, ball position and velocity, but not the configuration of the bricks, while available actions are just to move left or right. In Sapientino and Minecraft, the agent only knows its position in the grid (but not the color of the cells for Sapientino or the presence of resources or tools for Minecraft) and actions are one-step movements on the grid.

The plots in Figure 12.2 show learning performance in the three domains.

In the top row, we see an example of different reward machines applied to the same agent. The Breakout agent can move in the environment to intercept a ball and break the bricks in the environment. The two reward machines differ in the plan they represent: in the first case (plot in the left), the plan is to break four columns from left to right, in the second case (plot in the right), the plan is to break the columns from right to left.

In the middle row we show an example of using the same reward machine on two different agents: the reward machine drives the agents to learn a policy in which cells

of the grid are visited in a specified order, the two Sapientino agents differ in the state representations and actions available. In particular, the first agent (plot on the left) is an omni-directional robot moving in the four cardinal directions, while the second agent (plot on the right) is a differential drive robot moving forward/backward and turning left/right. The second agent includes also orientation as state variable. As shown in the figure, both agents are able to learn a policy according to the same reward machine, although with different low-level capabilities.

Finally, the bottom row shows the same reward machine applied to different Minecraft agents: omni-directional and differential drive. The state representation of the RL agent for Minecraft is exactly the same as the one used in Sapientino, while the set of actions are different in the two cases. Notice that an agent with the same state representation can learn Sapientino or Minecraft tasks only based on information received by the reward machine without requiring to change its state representations.

In all the situations, the RL agent learns a policy that achieve the goal as specified by the planning module. Moreover, learning is improved when using automatic sub task decomposition (blue curves) with respect to the standard RL algorithm (red curves).

These experimental results thus show convergence to a policy reaching the goal and improved performance when using sub task decomposition. Moreover, the results have been obtained in a modular way, as results reported in left and right plots have been obtained by just composing different instances of reward machine and RL agent, without changing their internal representations.

Finally, consider that, when explicit mapping is required, any combinatin of $\mathcal{RM}$ and RL agent requires a specific modeling effort. Thus, to execute the 6 experiments reported above, a total of 12 (6 $\mathcal{RM}$ + 6 RL) components must be devised. Our approach, instead, allows for re-using and combining existing components, without any additional modeling effort. Specifically, wrt the 6 experiments above, we have defined only 3 RL agents (1 for Breakout and 2 for both Sapientino and Minecraft) and 4 $\mathcal{RM}$ (2 for Breakout and 1 for each other problem), for a total of 7 components.

12.6 Summary and Discussion

Integration of planning and learning can benefit from modularity and separation of design and implementation of the relative components. In this chapter, we have shown that state and action representations of the two layers can be kept completely separated and only domain-independent signals are needed to ensure to drive the learning process through a desired plan to reach a given goal.

Future work includes extension of the formalism to more complex forms of plans, such a partial order plans, hierarchical task networks, conditional plans, and plans represented through Petri nets that would allow to generate compact reward machines for complex tasks.

We believe that design and development of modular planning and learning components will be convenient in many application domains, making the integration of planning and learning easier and more effective.

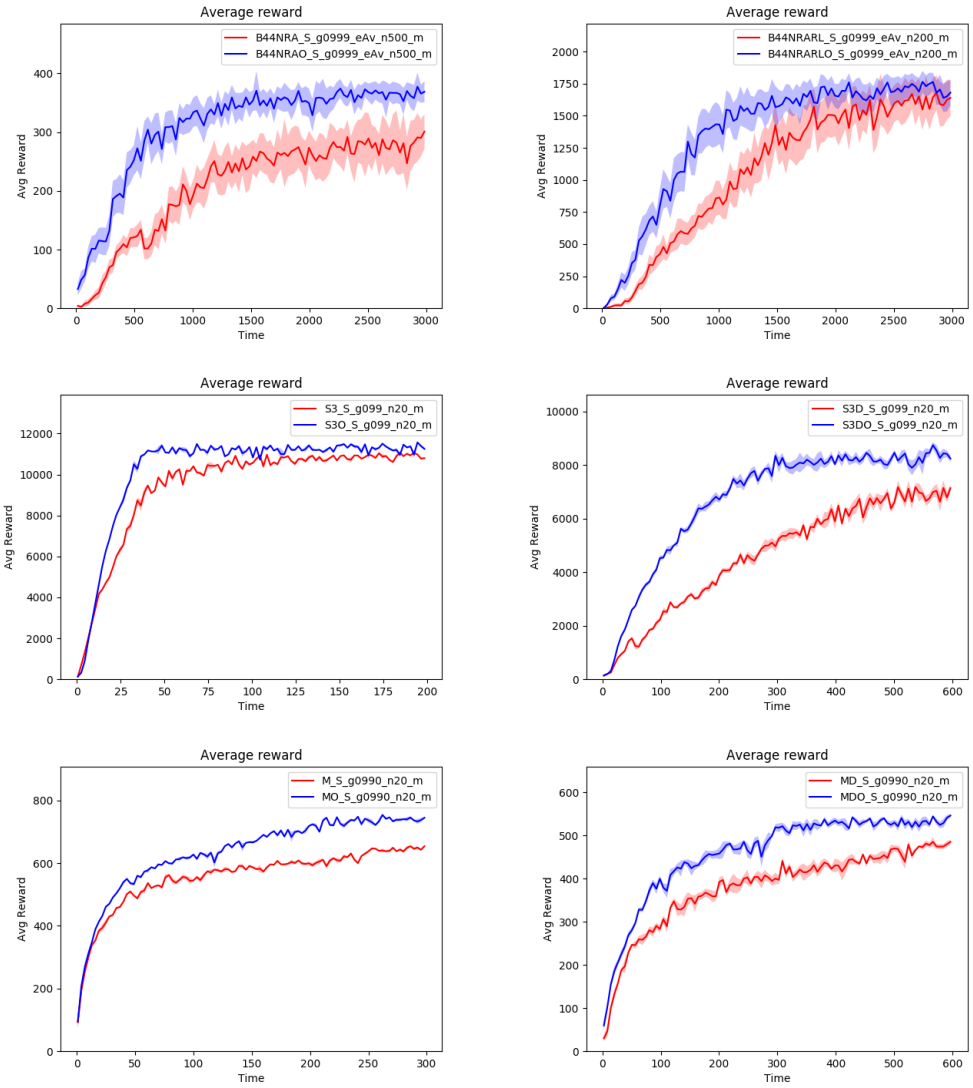


Figure 12.2. Average reward over experimental time. Breakout (top), Sapientino (middle), Minecraft (bottom), with sub task decomposition (blue), without sub task decomposition (red).

Part IV

Forward LTL_f Synthesis

Chapter 13

Background on LTL$_f$ Synthesis

This chapter gives background knowledge required for the contribution of Forward LTL$_f$ Synthesis, that will presented in Chapter 14.

- In Section 13.1, we revise linear temporal logic on finite traces, but with a variant of the syntax that (i) supports empty traces, and (ii) generates formulas in Negation Normal Form (i.e. negation only in front of propositional symbols).
- In Section 13.2, we give the definition of the problem of LTL$_f$ synthesis, and provide a general automata-theoretic solution based on DFA games.
- In Section 13.3, we introduce the topic of AND-OR graph search, a generalization of the classical graph search in which the OR nodes behave like the search nodes in classical sarch, and the AND nodes require the agent to find a solution for all their children;
- In Section 13.4, we introduce Sentential Decision Diagrams (SDD), a knowledge compilation technique that generalizes Binary Decision Diagrams, and its reduced and compressed form achieves canonicity wrt a hierarchy of variables (rather than an order of variables, as in BDDs).
- Section 13.5 concludes the chapter.

The contents of this chapter have been published in the conference paper (De Giacomo, Favorito, Jianwen, et al., 2022).

13.1 LTL$_f$ Basics

In this section, differently from Chapter 3, we define a syntactic variant of LTL$_f$ which only generates NNF formulas (without loss of generality). We also give further definitions useful for the next chapters.

Syntax. We require LTL$_f$ formulas are in Negation Normal Form (NNF), i.e., negations only occur in front of atomic propositions):

$$\varphi ::= tt \mid ff \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \bullet\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{R}\, \varphi_2.$$

where tt is always true, ff is always false; $p \in \mathcal{P}$ is an *atom*, and $\neg p$ is a *negated atom* (a literal l is an atom or the negation of an atom); $\wedge$ (AND) and $\vee$ (OR) are

the Booelean connectives; and $\bigcirc$ (Next), $\bullet$ (Weak Next), $\mathcal{U}$ (Until) and $\mathcal{R}$ (Release) are temporal connectives. We use the abbreviations $\mathit{true} := p \vee \neg p$, $\mathit{false} := p \wedge \neg p$, $\Diamond\varphi := \mathit{true}\,\mathcal{U}\,\varphi$ and $\Box\varphi := \mathit{false}\,\mathcal{R}\,\varphi$. Also for convenience we consider traces $\pi \in (2^{\mathcal{P}})^*$, that is we consider also empty traces ϵ as in (Brafman, De Giacomo, and Patrizi, 2018). We have that $\epsilon \models \varphi$ if φ is tt, a $\mathcal{R}$-formula or $\bullet$-formula, hence $\epsilon \models \Box \mathit{false}$. $\epsilon \not\models \varphi$ if φ is ff, an atom p or its negation $\neg p$, $\mathcal{U}$-formula or $\bigcirc$-formula, hence $\epsilon \not\models \Diamond \mathit{true}$.

A *trace* $\pi = \pi[0], \pi[1], \ldots$ is a sequence, possibly empty, of propositional interpretations (sets), in which $\pi[m] \in 2^{\mathcal{P}}$ $(0 \leq m < |\pi|)$ is the m-th interpretation of π, and $|\pi|$ represents the length of π. Intuitively, $\pi[m]$ is interpreted as the set of propositions which are *true* at instant m. We denote by $\pi(i,j) = \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}$, the segment of the trace π starting at the i-th step and ending at the$j$-th step (excluded). If $j > |\pi|$ then $\pi(i,j) = \pi(i,|\pi|)$. For every $j \leq i$, we have $\pi(i,j) = \epsilon$, i.e., the empty trace. Trace π is an *infinite* trace if $|\pi| = \infty$, which is formally denoted as $\pi \in (2^{\mathcal{P}})^{\omega}$; otherwise π is a *finite* trace, denoted as $\pi \in (2^{\mathcal{P}})^*$. LTL$_f$ formulas are interpreted over finite traces. Given a finite trace π and an LTL$_f$ formula φ, we inductively define when φ is *true* for π at point i $(0 \leq i < |\pi|)$, written $\pi, i \models \varphi$, as follows:

- $\pi, i \models tt$ and $\pi, i \not\models ff$;
- $\pi, i \models p$ iff $p \in \pi[i]$;
- $\pi, i \models \neg p$ iff $p \notin \pi[i]$;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \varphi_1 \vee \varphi_2$ iff $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$;
- $\pi, i \models \bigcirc\varphi$ iff $0 \leq i < |\pi| - 1$ and $\pi, i+1 \models \varphi$;
- $\pi, i \models \bullet\varphi$ iff $0 \leq i < |\pi|$ implies $\pi, i+1 \models \varphi$;
- $\pi, i \models \varphi_1\,\mathcal{U}\,\varphi_2$ iff there exists j with $i \leq j < |\pi|$ such that $\pi, j \models \varphi_2$, and for all k such that $i \leq k < j$ we have $\pi, k \models \varphi_1$;
- $\pi, i \models \varphi_1\,\mathcal{R}\,\varphi_2$ iff for all j with $i \leq j < |\pi|$ either we have $\pi, j \models \varphi_2$, or for some k such that $i \leq k < j$ we have $\pi, k \models \varphi_1$.

Moreover, for $i \geq |\pi|$, hence e.g., for $\pi = \epsilon$ we get:

- $\pi, i \models tt$;
- $\pi, i \not\models ff$;
- $\pi, i \not\models \varphi$ if φ is a literal;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \varphi_1 \vee \varphi_2$ iff $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$;
- $\pi, i \not\models \bigcirc\varphi$;
- $\pi, i \models \bullet\varphi$;
- $\pi, i \not\models \varphi_1\,\mathcal{U}\,\varphi_2$;

- $\pi, i \models \varphi_1 \mathcal{R} \varphi_2$.

An LTL$_f$ formula φ is *true* for π, denoted by $\pi \models \varphi$, if and only if $\pi, 0 \models \varphi$. In particular, we have $\epsilon \models \Box false$ and $\epsilon \not\models \Diamond true$. The set of finite traces that satisfy LTL$_f$ formula φ is the *language* of φ, denoted as $\mathcal{L}(\varphi) = \{\pi \in (2^{\mathcal{P}})^* \mid \pi \models \varphi\}$.

We denote by $\mathsf{cl}(\varphi)$ the set of subformulas of φ, including *tt* and *ff*. We denote by $\mathsf{pa}(\varphi) \subseteq \mathsf{cl}(\varphi)$ the set of literals and temporal subformulas of φ whose primary connective is temporal (Li, Rozier, et al., 2019). Formally, for an LTL$_f$ formula φ in NNF, we have $\mathsf{pa}(\varphi) = \{\varphi\}$ if φ is a literal or temporal formula; and $\mathsf{pa}(\varphi) = \mathsf{pa}(\varphi_1) \cup \mathsf{pa}(\varphi_2)$ if $\varphi = (\varphi_1 \wedge \varphi_2)$ or $\varphi = (\varphi_1 \vee \varphi_2)$. Having LTL$_f$ formula φ, replacing every temporal formula $\psi \in \mathsf{pa}(\varphi)$ with a propositional variable a_ψ gives us a propositional formula φ^p. As a consequence, two formulas φ_1 and φ_2 are propositionally equivalent, denoted by $\varphi_1 \sim_p \varphi_2$, if, $C \models \varphi_1^p \leftrightarrow C \models \varphi_2^p$ holds for every propositional assignment $C \in 2^{\mathsf{pa}(\varphi_1) \cup \mathsf{pa}(\varphi_2)}$. The equivalence class of a formula $\psi \in \mathsf{cl}(\varphi)$ is denoted by $[\psi]_{\sim_p}$ and defined as $[\psi]_{\sim_p} = \{y \in \mathsf{cl}(\varphi) \mid \psi \sim_p y\}$. The quotient set of a subset $C \subseteq \mathsf{cl}(\varphi)$ is denoted by $C/_{\sim_p}$ and defined as $C/_{\sim_p} = \{[\psi]_{\sim_p} \mid \psi \in C\}$.

Next Normal Form (XNF) Here we introduce the *neXt Normal Form* (XNF) (Li, Rozier, et al., 2019) of an LTL$_f$ formula, that will be useful later in the chapter. Intuitively, the XNF of an LTL$_f$ formula separates the "current-timestep" and "next-timestep" parts of the formula.

Definition 13.1. *A formula φ is in neXt Normal Form (XNF) if $\mathsf{pa}(\varphi)$ only includes literals, $\bigcirc$- and $\bullet$-formulas.*

For example, $\varphi = (a \mathcal{U} b)$ is not in XNF, while $(b \vee (a \wedge (X(a \mathcal{U} b))))$ is in XNF. Note that XNF transformation rules presented in prior works (Li, Rozier, et al., 2019; Xiao et al., 2021) does apply to full-fledged LTL$_f$. We thus present complete rules here. For an LTL$_f$ formula φ in NNF, the transformation function $\mathsf{xnf}(\varphi)$ is defined as follows:

- $\mathsf{xnf}(\varphi) = \varphi$ if φ is a literal, $\Box false$, $\Diamond true$, $\bigcirc$-formula, or $\bullet$-formula;
- $\mathsf{xnf}(\varphi_1 \wedge \varphi_2) = \mathsf{xnf}(\varphi_1) \wedge \mathsf{xnf}(\varphi_2)$;
- $\mathsf{xnf}(\varphi_1 \vee \varphi_2) = \mathsf{xnf}(\varphi_1) \vee \mathsf{xnf}(\varphi_2)$;
- $\mathsf{xnf}(\varphi_1 \mathcal{U} \varphi_2) = (\mathsf{xnf}(\varphi_2) \wedge \Diamond true) \vee (\mathsf{xnf}(\varphi_1) \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2))$;
- $\mathsf{xnf}(\varphi_1 \mathcal{R} \varphi_2) = (\mathsf{xnf}(\varphi_2) \vee \Box false) \wedge (\mathsf{xnf}(\varphi_1) \vee \bullet(\varphi_1 \mathcal{R} \varphi_2))$.

Theorem 13.2 ((Li, Rozier, et al., 2019))**.** *Every LTL$_f$ formula φ in NNF can be converted, with linear time in the formula size, to an equivalent formula in XNF, denoted by $\mathsf{xnf}(\varphi)$.*

13.2 LTL$_f$ Synthesis

The classical synthesis problem (Church, 1963; Vardi, 1995) consists in automatically constructing systems from logical specifications, such as formulas of a temporal logic. Because synthesis eliminates the need for a manual implementation, it has the potential to revolutionize the development process for reactive systems. And indeed, synthesis has, over the past few years, found applications in several areas of systems engineering, notably in the construction of circuits and device drivers and in the synthesis of controllers for robots and manufacturing plants.

In the infinite setting, reactive synthesis has been thoroughly investigated, starting from (Pnueli and Rosner, 1989). Unfortunately, while theoretically well investigated, algorithmically it still appears to be prohibitive in the infinite setting, not so much due to the high complexity of the problem, which is 2EXPTIME-complete, but for the difficulties of finding good algorithms for automata determinization, a crucial step in the solution, see, e.g., (Fogarty et al., 2015).

In the finite setting, the problem has been studied in (De Giacomo and Vardi, 2015). One of the best advantages is that the difficulties of automta determinization disappear and hence a theoretical solution to this problem actually promises to be appealing for effective practical implementation.

The synthesis problem is described follows. The set $\mathcal{P}$ of propositions into two disjoint sets $\mathcal{X}$ and $\mathcal{Y}$. We assume to have no control on the truth value of the propositions in $\mathcal{X}$, while we can control those in $\mathcal{Y}$. The problem then becomes: can we control the values of $\mathcal{Y}$ in such a way that for all possible values of $\mathcal{X}$ a certain LTL$_f$/LDL$_f$ formula φ remains true? More precisely, traces now assume the form $\pi = (X_0, Y_0)(X_1, Y_1)(X_2, Y_2) \cdots (X_n, Y_n)$, where (X_i, Y_i) is the propositional interpretation at the i-th position in π, now partitioned in the propositional interpretation X_i for $\mathcal{X}$ and Y_i for $\mathcal{Y}$. Let us denote by $\pi_{\mathcal{X}} \mid_u$ the interpretation π projected only on $\mathcal{X}$ and truncated at the i-th element (included), i.e., $\pi_X \mid i = X_0 X_1 \cdots X_i$. The realizability problem checks the existence of a function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$ such that for all π with $Y_i = f(\pi_{\mathcal{X}} \mid_i)$, we have that π satisfies the formula φ. The synthesis problem consists of actually computing such a function. Observe that in realizability/synthesis we have no way of constraining the value assumed by the propositions in $\mathcal{X}$: the function we are looking for only acts on propositions in $\mathcal{Y}$. Realizability and synthesis for LTL on infinite traces has been introduced in (Pnueli and Rosner, 1989) and shown to be 2EXPTIME-complete for arbitrary LTL formulas

DFA Games. DFA games are games between two players, here called respectively the environment and the controller, that are specified by a DFA. We have a set of $\mathcal{X}$ of *uncontrollable* propositions, which are under the control of the environment, and a set $\mathcal{Y}$ of *controllable* propositions, which are under the control of the controller. A *round* of the game consists of both the controller and the environment setting the values of the propositions they control. A (complete) *play* is a word in $(2^{\mathcal{X} \cup \mathcal{Y}})^*$ describing how the controller and environment set their propositions at each round till the game stops. The *specification* of the game is given by a DFA G of the form $G = \langle Q, 2^{\mathcal{X} \times \mathcal{Y}}, q_0, \delta, F \rangle$, where:

- Q are the states of the game;
- $2^{\mathcal{X} \times \mathcal{Y}}$ is the alphabet of the game;
- q_0 is the initial state of the game;
- $\delta : Q \times 2^{\mathcal{X} \cup \mathcal{Y}} \rightarrow Q$ is the transition function of the game: given the current state s and a choice of propositions X and Y , respectively for the enviroment and the controller, $\delta(q, (X, Y)) = q'$ is the resulting state of the game;
- F are the final states of the game, where the game can be considered terminated.

A play is *winning* for the controller if such a play leads from the initial to a final state. A *strategy* for the controller is a function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$ that, given a history of choices from the environment, decides which propositions Y to set to true/false next. A *winning strategy* is a strategy $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$ such that for all

π with $Y_i = f(\pi_{\mathcal{X}} \mid i)$ we have that π leads to a final state of G. The *realizability* problem consists of checking whether there exists a winning strategy. The *synthesis* problem amounts to actually computing such a strategy.

We now give a sound and complete technique to solve realizability for DFA games. We start by defining the *controllable preimage* $PreC(\mathcal{E})$ of a set $\mathcal{E}$ of states of $\mathcal{G}$ as the set of states q such that there exists a choice of values for propositions Y such that for all choices of values for propositions X, game $\mathcal{G}$ progresses to states in $\mathcal{E}$. Formally:

$$PreC(\mathcal{E}) = q \in Q \mid \exists Y \in 2^{\mathcal{Y}}.\forall X \in 2^{\mathcal{X}}.\delta(q,(X,Y)) \in \mathcal{E}$$

Using such a notion, we define the set $Win(\mathcal{G})$ of winning states of a DFA game $\mathcal{G}$, i.e., the set formed by the states from which the controller can win the DFA game $\mathcal{G}$. Specifically, we define $Win(\mathcal{G})$ as a least-fixpoint, making use of approximates $Win_i(\mathcal{G})$ denoting all states where the controller wins in at most i steps:

- $Win_0(\mathcal{G}) = F$ (the final states of $\mathcal{G}$);
- $Win_{i+1}(\mathcal{G}) = Win_i(\mathcal{G}) \cup PreC(Win_i(\mathcal{G}))$.

Then, $Win(\mathcal{G}) = \bigcup_i Win_i(\mathcal{G})$. Notice that computing $Win(\mathcal{G})$ requires linear time in the number of states in $\mathcal{G}$. Indeed, after at most a linear number of steps $Win_{i+1}(\mathcal{G}) = Win_i(\mathcal{G}) = Win(\mathcal{G})$.

Theorem 13.3 ((De Giacomo and Vardi, 2015))**.** *A* DFA *game* $\mathcal{G}$ *admits a winning strategy iff* $q_0 \in Win(\mathcal{G})$.

Next, we turn to actually computing the strategy. To do so, we define a *strategy generator* based on the winning sets $Win_i(\mathcal{G})$. This is a nondeterministic transducer, where nondeterminism is of the kind "don't-care": all nondeterministic choices are equally good. The strategy generator $T_{\mathcal{G}} = \langle Q, 2^{\mathcal{X}\times\mathcal{Y}}, q_0, \delta, \omega\rangle$ is as follows:

- Q are the states of the transducer;
- $2^{\mathcal{X}\times\mathcal{Y}}$ is the alphabet of the transducer;
- q_0 is the initial state;
- $\delta : Q \times 2^{\mathcal{X}} \to 2^Q$ is the transition function such that $\delta(q,X) = \{q' \mid q' = \delta(q,(X,Y))$ and $Y \in \omega(q)\}$;
- $\omega : Q \to 2^{\mathcal{Y}}$ is the output function such that $\omega(q) = \{Y \mid$ if $q \in Win_{i+1}(\mathcal{G}) - Win_i(\mathcal{G})$ then $\forall X.\delta(q,(X,Y)) \in Win_i(\mathcal{G})\}$.

The transducer $T_{\mathcal{G}}$ generates strategies in the following sense: for every way of further restricting $\omega(q)$ to return only one of its values (chosen arbitrarily), we get a strategy.

Automata-based Synthesis in LTLf and LDLf. To do synthesis in LTL$_f$ or LDL$_f$, we translate an LTL$_f$/LDL$_f$ specification φ into an AFA A_φ, as explained in Chapter 4. Then, we determinize to get a DFA $\mathcal{A}^d_\varphi$. This will cost us two exponentials. At this point we view the resulting DFA A^d_φ as a DFA game, considering exactly the separation between controllable and uncontrollable propositions in the original LTL$_f$/LDL$_f$ specification, and we solve it by computing $Win(A^d_\varphi)$ and the corresponding strategy generator $\mathcal{T}_{A^d_\varphi}$. This is a linear step.

Theorem 13.4 ((De Giacomo and Vardi, 2015))**.** *Realizability (and synthesis) in* LTL$_f$/LDL$_f$ *is 2EXPTIME-hard.*

13.3 AND-OR Graph Search

Being a popular topic in AI, AND-OR graph search has attracted extensive studies. Following (Nilsson, 1971), an AND-OR graph can be considered as a generalization of a directed graph, where there are a set of nodes $\mathcal{V}$ and generalized connectors (edges) between nodes. Every connector links one single node $v \in \mathcal{V}$ to a set of nodes $V \subseteq \mathcal{V}$, where n is the number of nodes in the graph. A connector is called an AND (resp. OR) connector, if there is a logical AND (resp. OR) relationship among V. It should be noted that in this work we only focus on specific AND-OR graphs, where every node has only one connector leading to its successor nodes. Therefore, we have AND-nodes with an AND connector, and OR-nodes with an OR connector. Moreover, the set of goal nodes V_g only consists of OR-nodes.

The AND-OR graph search problem was first introduced in (Nilsson, 1971). Intuitively speaking, the searching procedure aims to find a winning plan that encodes a path leading from the initial node to goal nodes. It is possible to involve both kinds of nodes in the winning plan, therefore, the plan lists one outgoing option at OR-nodes, and all outgoing options at AND-nodes leading to branches. Therefore, a winning plan is essentially a tree such that all leaves are goal nodes. There has been extensive studies on AND-OR graph search techniques (Mahanti and Bagchi, 1985; Jiménez and Torras, 2000), and have been utilized in a lot of applications, e.g., FOND planning (Mattmüller et al., 2010; Mattmüller, 2013; Geffner and Bonet, 2013).

13.4 Sentential Decision Diagrams (SDDs)

SDDs (Darwiche, 2011) is a Knowledge Compilation (KC) technique designed for an efficient representation and manipulation of Boolean functions. In order to represent a Boolean function, the classical method is applying Shannon decomposition, as done in Ordered Binary Decision Diagrams (BDDs) (Bryant, 1992). Intuitively, BDD decomposes Boolean functions with one variable at a time. Let f be a Boolean function over $\mathcal{Y} \cup \mathcal{X}$, where $\mathcal{Y} = \{y_1, y_2\}$ and $\mathcal{X} = \{x_1, x_2\}$ such that $\mathcal{Y}, \mathcal{X}$ are non-intersecting sets of variables. BDD considers $\mathcal{Y} \cup \mathcal{X}$ as a single set $\{y_1, x_1, y_2, x_2\}$, instead of an $(\mathcal{Y}, \mathcal{X})$-partition, such that obtaining $(y_1, f|y_1), (\neg y_1, f(\neg y_1))$ in the first decomposition, where y_1 and $\neg y_1$ are disjoint. In the next round of decomposition, we get $(y_1 \wedge x_1, f|(y_1 \wedge x_1)), (y_1 \wedge \neg x_1, f|(y_1 \wedge \neg x_1)), (\neg y_1 \wedge x_1, f(\neg y_1, x_1)), (\neg y_1 \wedge \neg x_1, f|(\neg y_1 \wedge \neg x_1))$ and so on. Therefore, the canonicity of BDD is determined referring to a specific ordering of variables, that the decomposition procedure follows.

SDD, instead, utilizes a more general decomposition that decomposes a set of variables at each round. Given an $(\mathcal{Y}, \mathcal{X})$-partition, where $\mathcal{Y}$ variables are considered *primary* and $\mathcal{X}$ variables are considered *subsequent*. The SDD of f, with respect to the $(\mathcal{Y}, \mathcal{X})$-partition, can be written as $\bigvee_{i=1}^{n}[\mathsf{prime}_i(\mathcal{Y}) \wedge \mathsf{sub}_i(\mathcal{X})]$. Intuitively, SDD decomposes f into n children, each of which consists of Boolean functions $\mathsf{prime}_i(\mathcal{Y})$ (what are satisfied *in primary*) and $\mathsf{sub}_i(\mathcal{X})$ (what should be satisfied *in subsequent*, according to $\mathsf{prime}_i(\mathcal{Y})$). In particular, besides that all the primes are disjoint and covering, i.e., $\mathsf{prime}_i \wedge \mathsf{prime}_j = \mathit{false}$ for $i \neq j$, and $\bigvee_{i=1}^{n} \mathsf{prime}_i = \mathit{true}$, SDD also guarantees that all the subs are compressed, i.e., $\mathsf{sub}_i(\mathcal{X}) \neq \mathsf{sub}_j(\mathcal{X})$ for $i \neq j$. Hence, the canonicity of SDDs is determined wrt a specific partition of variables, represented as an ordered full binary tree, called *vtree*, as shown in Figure 13.1 [1].

[1] Example is from (Darwiche, 2011)

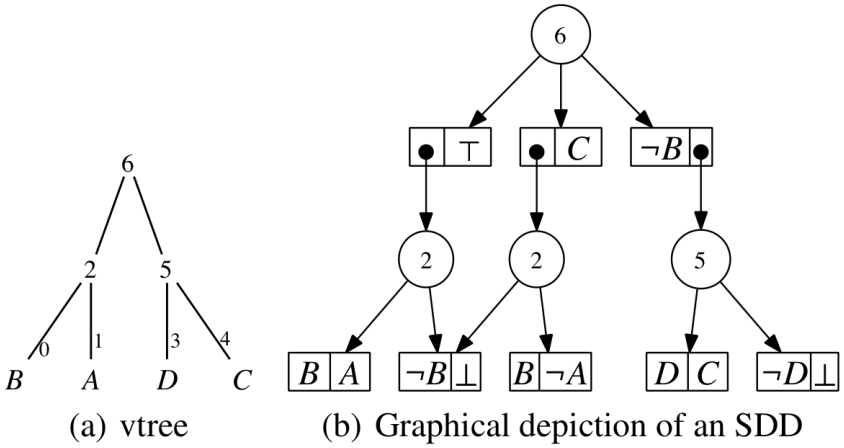


Figure 13.1. Function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

The leaves of a *vtree* have a one-to-one correspondence with variables. Here, each internal node partitions the variables into those in the left subtree ($\mathcal{Y}$) and those in the right subtree ($\mathcal{X}$). SDDs, similarly to BDDs, can be reduced to a canonical representation by merging identical substructures. In particular, given a *vtree*, for any Boolean function f, there is a unique reduced SDD that represents f (Darwiche, 2011).

13.5 Summary

This section provided preliminary knowledge for the following chapter on forward LTL$_f$ synthesis. We first introduced a version of LTL$_f$ that also supports empty traces and it is only in Negation Normal Form. Then, we presented the standard definition of LTL$_f$ synthesis, and a solution based on DFA games. We then presented the topic of AND-OR graph search, a variant of graph search suitable for LTL$_f$ synthesis, as we shall see in the next chapter, and Sentiential Decision Diagrams, a knowledge compilation technique that will be useful in our forward LTL$_f$ synthesis approach.

Chapter 14

LTL$_f$ Synthesis as AND-OR Graph Search

Synthesis techniques for temporal logic specifications are typically based on exploiting symbolic techniques, as done in model checking. These symbolic techniques typically use backward fixpoint computation. Planning, which can be seen as a specific form of synthesis, is a witness of the success of forward search approaches. In this chapter, we develop a forward-search approach to full-fledged Linear Temporal Logic on finite traces (LTL$_f$) synthesis. We show how to compute the Deterministic Finite Automaton (DFA) of an LTL$_f$ formula on-the-fly, while performing an adversarial forward search towards the final states, by considering the DFA as a sort of AND-OR graph. Our approach is characterized by branching on suitable propositional formulas, instead of individual evaluations, hence radically reducing the branching factor of the search space. Specifically, we take advantage of techniques developed for knowledge compilation, such as Sentential Decision Diagrams (SDDs), to implement the approach efficiently.

The rest of the chapter is structured as follows:

- In Section 14.1, we introduce the problem and give the motivations for this work.
- In Section 14.2 we introduce a novel technique to build a DFA based on LTL$_f$ formula progression.
- In Section 14.3, we reduce the problem of LTL$_f$ synthesis to AND-OR graph search. We propose an abstract depth-first algorithm that can be instantiated depending on how the successor transitions are computed from a search node (the EXPAND "abstract" function).
- In Section 14.4, we review related work, highlighting the differences with our approach.
- Section 14.5 concludes the chapter, discusses the results and give insights on potential future works.

The contents of this chapter have been published in the conference paper (De Giacomo, Favorito, Jianwen, et al., 2022).

14.1 Introduction

Program synthesis aims at automatically generating a program from declarative specifications expressed in temporal logic (Pnueli and Rosner, 1989; Ehlers et al., 2017). A commonly used logic for program synthesis is Linear Temporal Logic (LTL), typically used also in model checking (Baier and Katoen, 2008). Recently, synthesis has been investigated for specifications expressed in LTL_f, a finite-trace variant of LTL (De Giacomo and Vardi, 2013). Roughly speaking, we consider an alphabet of propositions partitioned into those controlled by the agent (one may think of these as a binary encoding of agent actions) and those controlled by the environment (one may think of these as fluents), and then we use LTL_f to specify which finite traces are desirable. The outcome of the synthesis procedure is a program (a finite-state controller) that at every time step, given the values of the environment propositions in the history so far, sets the next value of the agent propositions so that the traces generated satisfy the LTL_f specification (De Giacomo and Vardi, 2015).

LTL_f synthesis has been proven to be one of the most successful synthesis settings so far. Several tools have been developed recently, among which Lisa (Bansal et al., 2020) and Lydia (De Giacomo and Favorito, 2021) are possibly the best performing ones to date. Both these tools are based on first constructing a DFA corresponding to the LTL_f specification, and then considering it as a game arena where the agent tries to get to an accepting state in spite that the environment tries to avoid it. A winning strategy, which is a finite controller returned by the synthesis procedure, can be obtained through a backward fixpoint computation for *adversarial reachability* of the DFA accepting states (De Giacomo and Vardi, 2015). The main difficulty of this approach is that it requires computing the entire DFA of the LTL_f specification, which can be, in the worst case, doubly exponential in the size of the specification (De Giacomo and Vardi, 2015). Hence, even though the backward fixpoint computation can be performed symbolically, enabling scalable performance (Zhu, Tabajara, Li, et al., 2017), the DFA construction step can become a significant bottleneck (Zhu, Pu, and Vardi, 2019).

An alternative approach is to expand the arena while searching for the accepting states via forward search (Xiao et al., 2021), which is analogous to the approach taken by most work in adversarial Planning with fully observable nondeterministic domains (FOND), where the agent controls the actions and the environment controls the fluents (Ghallab, S. Nau, and Traverso, 2004; Geffner and Bonet, 2013). The agent has to reach the goal, despite that the environment may choose adversarially the effects of the agent actions (*strong plans* in FOND) (Cimatti, Roveri, and Traverso, 1998; Cimatti, Pistore, et al., 2003; Geffner and Bonet, 2013). The typical way to deal with this kind of planning is through forward search on an AND-OR graph (Nilsson, 1971), where the OR-nodes correspond to the choices (quantified existentially) of the agent and the AND-nodes correspond to the choices (quantified universally) of the environment (Mattmüller et al., 2010; Mattmüller, 2013; Geffner and Bonet, 2013). Note that the search space generated for FOND planning with a compactly represented domain, say, in PDDL (Haslum et al., 2019), is at most single-exponential (Rintanen, 2004).

Instead, to handle LTL_f synthesis, we need to deal with a state space that can be of double-exponential size. Searching over a double-exponential state space has been studied in Planning in partially observable nondeterministic domains (POND), aka *contingent planning*, where the search procedure must be performed over the *belief-states* (Reif, 1984; Goldman and Boddy, 1996; Bertoli et al., 2006; Geffner and Bonet, 2013). However, belief-states have a specific structure (Bertoli et al., 2006; Thanh To, Pontelli, and Cao Son, 2009), the techniques utilized in contingent

planning cannot be directly applied to LTL$_f$ synthesis.

In this work, we investigate LTL$_f$ forward synthesis adopting an AND-OR graph search as in FOND Planning (Mattmüller et al., 2010; Mattmüller, 2013), but over a doubly exponential search space, as for contingent planning (Bertoli et al., 2006). We do not rely on an encoding into PDDL, as (Camacho, Baier, et al., 2018; Camacho and McIlraith, 2019b), which may result into a PDDL specification with exponential size. Instead, we develop specific techniques to create the search space on-the-fly while exploring it, such that we can possibly decide realizability/unrealizability before reaching the worst-case double-exponential blowup.

In details, we propose a technique to create on-the-fly the DFA corresponding to the LTL$_f$ specification. This technique avoids a detour to automata theory and instead builds directly deterministic transitions from a current state. In particular, this technique exploits LTL formula progression (Emerson, 1990; Bacchus and Kabanza, 1998) to separate what happens *now* (label) and what should happen *next* accordingly (successor state). Crucially, we exploit the structure that formula progression provides to branch on propositional formulas (representing several evaluations), instead of individual evaluations. This drastically reduces the branching factor of the AND-OR graph to be searched (recall that in LTL$_f$ synthesis, both the agent choices and the environment choices can be exponentially many). More specifically, we label transitions/edges with propositional formulas on propositions controlled by the agent (for OR-nodes) and by the environment (for AND-nodes). Every such propositional formula captures a set of evaluations leading to the same successor node. We leverage Knowledge Compilation (KC) techniques, and in particular Sentential Decision Diagrams (SDDs) (Darwiche, 2011), to effectively generate such propositional formulas for OR-nodes and AND-nodes, and thus reduce the branching factor of the search space. We implemented our approach in a tool called Cynthia and conducted comprehensive experiments by comparing to existing LTL$_f$ synthesis tools, including Lisa, Lydia and Ltlfsyn from (Xiao et al., 2021) and demonstrate the merits of our approach.

14.2 DFA Construction from LTL$_f$

The classical approach to LTL$_f$ synthesis first constructs the complete DFA, and then solves an adversarial reachability game through a backward fixpoint computation on this DFA (De Giacomo and Vardi, 2015). An alternative approach presented in (Xiao et al., 2021) is an on-the-fly synthesis technique that is able to construct the automaton while solving the game in a forward way. Yet, the game arena generated there is explicit, s.t. during search, there can be an exponential number of options to explore at every state, leading to a major drawback for scalability. We now present a new DFA construction based on an incremental technique called *formula progression* that is suitable for exploiting SDDs.

LTL$_f$ Formula Progression. Consider an LTL$_f$ formula φ over $\mathcal{P}$ and a finite trace $\pi = \pi[0], \pi[1], \ldots \in (2^{\mathcal{P}})^*$, in order to have $\pi \models \varphi$, we can start from φ, progress or push φ through π. The idea behind *formula progression* is to consider LTL$_f$ formula φ into a requirement about *now* $\pi[i]$, which can be checked straightaway, and a requirement about the future that has to hold on the yet unavailable suffix. That is to say, formula progression looks at $\pi[i]$ and φ, and progresses a new formula $\mathsf{fp}(\varphi, \pi[i])$ such that $\pi, i \models \varphi$ iff $\pi, i+1 \models \mathsf{fp}(\varphi, \pi[i])$. This procedure is analogous to DFA reading trace π, where reaching accepting states is essentially achieved by taking one transition after another. Formula progression has been studied in prior work, cf. (Emerson, 1990; Bacchus and Kabanza, 1998). Here we use it for

constructing DFA from LTL_f formulas.

Note that, since π is a finite trace, it is necessary to clarify when the trace ends. To do so, we introduce two new formulas $\Box \mathit{false}$ and $\Diamond \mathit{true}$, which, intuitively, refer to *finite trace ends* and *finite trace not ends*, respectively. For simplicity, we enrich $\mathsf{cl}(\varphi)$, the set of proper subformulas of φ, to include them such that $\mathsf{cl}(\varphi)$ is reloaded as $\mathsf{cl}(\varphi) \cup \mathsf{cl}(\Diamond \mathit{true}) \cup \mathsf{cl}(\Box \mathit{false})$.

Definition 14.1 (LTL_f Formula Progression)**.** *For an LTL_f formula φ in NNF, the progression function $\mathsf{fp}(\varphi, \sigma)$, where $\sigma \in 2^{\mathcal{P}}$, is defined as follows:*

- $\mathsf{fp}(\mathit{tt}, \sigma) = \mathit{tt}$ *and* $\mathsf{fp}(\mathit{ff}, \sigma) = \mathit{ff}$*;*
- $\mathsf{fp}(p, \sigma) = \mathit{tt}$ *if* $p \in \sigma$*, otherwise* ff*;*
- $\mathsf{fp}(\neg p, \sigma) = \mathit{tt}$ *if* $p \notin \sigma$*, otherwise* ff*;*
- $\mathsf{fp}(\varphi_1 \wedge \varphi_2, \sigma) = \mathsf{fp}(\varphi_1, \sigma) \wedge \mathsf{fp}(\varphi_2, \sigma)$*;*
- $\mathsf{fp}(\varphi_1 \vee \varphi_2, \sigma) = \mathsf{fp}(\varphi_1, \sigma) \vee \mathsf{fp}(\varphi_2, \sigma)$*;*
- $\mathsf{fp}(\bigcirc \varphi, \sigma) = \varphi \wedge \Diamond \mathit{true}$*;*
- $\mathsf{fp}(\bullet \varphi, \sigma) = \varphi \vee \Box \mathit{false}$*;*
- $\mathsf{fp}(\varphi_1 \mathcal{U} \varphi_2, \sigma) = \mathsf{fp}(\varphi_2, \sigma) \vee (\mathsf{fp}(\varphi_1, \sigma) \wedge \mathsf{fp}(\bigcirc(\varphi_1 \mathcal{U} \varphi_2), \sigma))$*;*
- $\mathsf{fp}(\varphi_1 \mathcal{R} \varphi_2, \sigma) = \mathsf{fp}(\varphi_2, \sigma) \wedge (\mathsf{fp}(\varphi_1, \sigma) \vee \mathsf{fp}(\bullet(\varphi_1 \mathcal{R} \varphi_2), \sigma))$*.*

Note that $\mathsf{fp}(\varphi, \sigma)$ is a positive Boolean formula on $\mathsf{cl}(\varphi)$, i.e., $\mathsf{fp}(\varphi, \sigma) \in \mathcal{B}^+(\mathsf{cl}(\varphi))$.

The following two lemmas show that $\mathsf{fp}(\varphi, \sigma)$ strictly follows LTL_f semantics and retains the propositional behavior of LTL_f formulas.

Lemma 14.2. *Let φ be an LTL_f formula over $\mathcal{P}$ in NNF, π be a finite nonempty trace, $\mathsf{fp}(\varphi, \sigma)$ be as above. We have that $\pi, i \models \varphi$ iff $\pi, i+1 \models \mathsf{fp}(\varphi, \pi[i])$.*

Proof. We prove by structural induction on the formula.

- $\varphi = \mathit{tt}$. Note that $\mathsf{fp}(\mathit{tt}, \pi[i]) = \mathit{tt}$. Therefore, we have $\pi, i \models \mathit{tt}$ iff $\pi, i+1 \models \mathsf{fp}(\mathit{tt}, \pi[i]) = \mathit{tt}$.
- $\varphi = \mathit{ff}$. Note that $\mathsf{fp}(\mathit{ff}, \pi[i]) = \mathit{ff}$. Therefore, we have $\pi, i \models \mathit{ff}$ iff $\pi, i+1 \models \mathsf{fp}(\mathit{ff}, \pi[i]) = \mathit{ff}$.
- $\varphi = p$. Note that either $\mathsf{fp}(p, \pi[i]) = \mathit{tt}$ or $\mathsf{fp}(p, \pi[i]) = \mathit{false}$. We now have that $\pi, i \models p$ iff $p \in \pi[i]$ iff $\mathsf{fp}(p, \pi[i]) = \mathit{tt}$ iff $\pi, i+1 \models \mathit{tt}$.
- $\varphi = \neg p$. Note that either $\mathsf{fp}(\neg p, \pi[i]) = \mathit{tt}$ or $\mathsf{fp}(\neg p, \pi[i]) = \mathit{false}$ We have that $\pi, i \models \neg p$ iff $p \notin \pi[i]$ iff $\mathsf{fp}(\neg p, \pi[i]) = \mathit{tt}$ iff $\pi, i+1 \models \mathit{tt}$.
- $\varphi = \varphi_1 \wedge \varphi_2$. Note that $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$. By induction hypotheses, we have iff $\pi, i+1 \models \mathsf{fp}(\varphi_1, \pi[i])$ and $\pi, i+1 \models \mathsf{fp}(\varphi_2, \pi[i])$. By definition of fp, we have that iff $\pi, i+1 \models \mathsf{fp}(\varphi_1, \pi[i]) \wedge \mathsf{fp}(\varphi_2, \pi[i])$, and thus iff $\pi, i+1 \models \mathsf{fp}(\varphi_1 \wedge \varphi_2, \pi[i])$.
- $\varphi = \varphi_1 \vee \varphi_2$. Note that $\pi, i \models \varphi_1 \vee \varphi_2$ iff $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$. By induction hypotheses, we have iff $\pi, i+1 \models \mathsf{fp}(\varphi_1, \pi[i])$ or $\pi, i+1 \models \mathsf{fp}(\varphi_2, \pi[i])$. By definition of fp, we have that iff $\pi, i+1 \models \mathsf{fp}(\varphi_1, \pi[i]) \vee \mathsf{fp}(\varphi_2, \pi[i])$, and thus iff $\pi, i+1 \models \mathsf{fp}(\varphi_1 \vee \varphi_2, \pi[i])$.

- $\varphi = \mathsf{O}\psi$. Note that $\varphi = \mathsf{O}\psi$ iff $\pi, i+1 \models \psi$ and $i+1 < |\pi|$ iff $\pi, i+1 \models \psi \wedge \pi, i+1 \models \Diamond true$, since $\Diamond true \iff i+1 < |\pi|$, i.e. $\Diamond true$ requires another step to be read. By definition of fp, we have that iff $\pi, i+1 \models \mathsf{fp}(\mathsf{O}\psi, \pi[i])$.

- $\varphi = \bullet\psi$. Note that $\varphi = \bullet\psi$ iff $\pi, i+1 \models \psi$ or $\neg(i+1 < |\pi|)$ iff $\pi, i+1 \models \psi$ or $\pi, i+1 \models \Box false$, since $\Box false \iff \neg(i+1 < |\pi|)$, i.e. the trace is empty. By definition of fp, we have that iff $\pi, i+1 \models \mathsf{fp}(\bullet\psi, \pi[i])$.

- $\varphi = \varphi_1 \mathcal{U} \varphi_2$. Note that $\varphi = \varphi_1 \mathcal{U} \varphi_2$ iff $\pi, i \models \varphi_2 \vee (\varphi_1 \wedge \mathsf{O}(\varphi_1 \mathcal{U} \varphi_2))$ iff $\pi, i \models \varphi_2 \vee (\pi, i \models \varphi_1 \wedge \pi, i+1 \models \varphi_1 \mathcal{U} \varphi_2 \wedge \pi, i+1 \models \Diamond true)$ (note that the addition of the term with $\Diamond true$ does not change the meaning of the formula, as it is subsumed by $\varphi_1 \mathcal{U} \varphi_2$). By induction hypotheses, we have iff $\pi, i{+}1 \models \mathsf{fp}(\varphi_2, \pi[i]) \vee (\mathsf{fp}(\varphi_1, \pi[i]) \wedge \mathsf{fp}(\mathsf{O}(\varphi_1 \mathcal{U} \varphi_2), \pi[i]))$. By definition, we have iff $\pi, i{+}1 \models \mathsf{fp}(\varphi_2, \pi[i]) \vee (\pi, i{+}1 \models \mathsf{fp}(\varphi_1, \pi[i]) \wedge \pi, i{+}1 \models \varphi_1 \mathcal{U} \varphi_2 \wedge \pi, i{+}1 \models \Diamond true)$ iff $\pi, i+1 \models \mathsf{fp}(\varphi_1 \mathcal{U} \varphi_2, \pi[i])$.

- $\varphi = \varphi_1 \mathcal{R} \varphi_2$. Note that $\varphi = \varphi_1 \mathcal{R} \varphi_2$ iff $\pi, i \models \varphi_2 \wedge (\varphi_1 \vee \bullet(\varphi_1 \mathcal{R} \varphi_2))$ iff $\pi, i \models \varphi_2 \vee (\pi, i \models \varphi_1 \wedge \pi, i+1 \models \varphi_1 \mathcal{R} \varphi_2 \vee \pi, i+1 \models \Box false)$ (note that the addition of the term with $\Box false$ does not change the meaning of the formula, as it is subsumed by $\varphi_1 \mathcal{R} \varphi_2$). By induction hypotheses, we have iff $\pi, i+1 \models \mathsf{fp}(\varphi_2, \pi[i]) \wedge (\mathsf{fp}(\varphi_1, \pi[i]) \vee \mathsf{fp}(\bullet(\varphi_1 \mathcal{R} \varphi_2), \pi[i]))$. By definition, we have iff $\pi, i{+}1 \models \mathsf{fp}(\varphi_2, \pi[i]) \wedge (\pi, i{+}1 \models \mathsf{fp}(\varphi_1, \pi[i]) \vee \pi, i{+}1 \models \varphi_1 \mathcal{R} \varphi_2 \vee \pi, i+1 \models \Box false)$ iff $\pi, i+1 \models \mathsf{fp}(\varphi_1 \mathcal{R} \varphi_2, \pi[i])$. □

Lemma 14.3. *Let φ and ψ be two* LTL$_f$ *formulas over $\mathcal{P}$ in* NNF *s.t. $\varphi \sim_p \psi$, and $\sigma \in 2^{\mathcal{P}}$. Then $\mathsf{fp}(\varphi, \sigma) \sim_p \mathsf{fp}(\psi, \sigma)$ holds.*

Proof. Since φ and ψ are propositionally equivalent, the propositional semantics thus maps φ and ψ to the same monotone Boolean function over alphabet of $\mathsf{pa}(\varphi) \cup \mathsf{pa}(\psi)$. By definition, $\mathsf{fp}(\varphi, \sigma)$ can be considered as the result of replacing the occurrences of $\theta \in \mathsf{pa}(\varphi) \cup \mathsf{pa}(\psi)$ with $\mathsf{fp}(\theta, \sigma)$, without changing any disjunction or conjunction. This is the same for $\mathsf{fp}(\psi, \sigma)$. Therefore, fp preserves the propositional equivalence, and thus $\mathsf{fp}(\varphi, \sigma) \sim_p \mathsf{fp}(\psi, \sigma)$ holds. □

We generalize LTL$_f$ formula progression from single instants to finite traces by defining $\mathsf{fp}(\varphi, \epsilon) = \varphi$, and $\mathsf{fp}(\varphi, \sigma u) = \mathsf{fp}(\mathsf{fp}(\varphi, \sigma), u)$, where $\sigma \in 2^{\mathcal{P}}$ and $u \in (2^{\mathcal{P}})^*$.

Lemma 14.4. *Let φ be an* LTL$_f$ *formula over $\mathcal{P}$ in* NNF*, π be a finite trace. We have that $\pi \models \varphi$ iff $\epsilon \models \mathsf{fp}(\varphi, \pi)$.*

Proof. Let n denote the length of π such that $n = |\pi|$. From Lemma 14.2, we have that $\pi \models \varphi$ iff $\pi, 1 \models \mathsf{fp}(\varphi, \pi[0])$, moreover iff $\pi, n \models \mathsf{fp}(\mathsf{fp}(\varphi, \pi^{n-2}), \pi[n-1])$ is true. That is to say, iff $\epsilon \models \mathsf{fp}(\varphi, \pi)$ holds. □

Given an LTL$_f$ formula φ, we can consider it as the initial state, and recursively apply formula progression to obtain all reachable states (through deterministic transitions), denoted by $\mathrm{Reach}(\varphi) = \{\mathsf{fp}(\varphi, \pi) \mid \pi \in (2^{\mathcal{P}})^*\}$. Note that once applying propositional equivalence, there can only be $2^{2^{|\mathsf{cl}(\varphi)|}}$ elements in $\mathrm{Reach}(\varphi)/_{\sim_p}$. Lemma 14.4 shows that a state $\psi \in \mathrm{Reach}(\varphi)/_{\sim_p}$ can be recognized as accepting iff $\epsilon \models \psi$, indicating that there exists a trace π such that $\psi = \mathsf{fp}(\varphi, \pi)$ and π is completely "consumed" by formula progression, returning a formula, corresponding to an accepting state, that holds on the empty trace ϵ. In particular, given that every state ψ is actually a positive Boolean formula on $\mathsf{cl}(\varphi)$, checking $\epsilon \models \psi$ only requires dealing with Boolean operators of disjunction and conjunction, which can

be done in linear time. The correctness and complexity of our DFA construction are stated below.

Theorem 14.5. *Given* LTL$_f$ *formula* φ*, the following DFA recognizes* $\mathcal{L}(\varphi)$*:* $\mathcal{A} = (2^{\mathcal{P}}, S, s_0, \delta, Acc)$*, where the states* $S = \text{Reach}(\varphi)/_{\sim_p}$*, the initial state* $s_0 = \varphi_{\sim_p}$*, the transition function* $\delta([\psi]_{\sim_p}, \sigma) = \mathsf{fp}([\psi]_{\sim_p}, \sigma), \forall \sigma \in 2^{\mathcal{P}}$ *and the accepting states* $Acc = \{\psi \mid \epsilon \models \psi\}$.

Proof. In order to prove $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$, it is enough to show that every finite trace $\pi \models \varphi$ iff $\pi \in \mathcal{L}(\mathcal{A})$. Let $\pi \in (2^{\mathcal{P}})^*$ be a finite trace, and $|\pi| = n$ such that $n \geq 0$.

($\Rightarrow$) We perform this part of the proof in two cases over the value of n.

- If $n = 0$ (i.e., $\pi = \epsilon$), $\epsilon \models \varphi$ implies that the initial state $\varphi \in Acc$. So it holds that ϵ is accepted by $\mathcal{A}$, i.e., $\pi \in \mathcal{L}(\mathcal{A})$.
- If $n > 0$, there exists a run $r = s_0, \ldots, s_n$ over $\mathcal{A}$ on π such that $s_0 = \varphi$ is the initial and $\delta(s_i, \pi[i]) = s_{i+1}$ holds for $0 \leq i < n$. We first prove that $\pi, i \models s_i$ for $0 \leq i \leq n$ by induction over the value of i.
  Basis. $\pi \models \varphi$ i.e., $\pi, 0 \models s_0$ holds basically.
  Induction. The induction hypothesis is $\pi, i \models s_i$ holds where $0 \leq i < n$. By the induction hypothesis and Lemma 14.2 we can get $\pi, i+1 \models \mathsf{fp}(s_i, \pi[i])$.Note that $\mathsf{fp}(s_i, \pi[i]) = \delta(s_i, \pi[i]) = s_{i+1}$, so it holds that $\pi, i+1 \models s_{i+1}$.
  Therefore, we have $\pi, n \models s_n$. Since $\pi, n = \epsilon$, $s_n \in Acc$ is an accepting state. So it holds that π is accepted by $\mathcal{A}$, i.e., $\pi \in \mathcal{L}(\mathcal{A})$.

($\Leftarrow$) $\pi \in \mathcal{L}(\mathcal{A})$ implies that there exists a run $r = s_0, \ldots, s_n$ over $\mathcal{A}$ on π such that $\delta(s_i, \pi[i]) = s_{i+1}$ holds for $0 \leq i < n$ and $s_n \in Acc$.

We first prove that $\pi \in \mathcal{L}(\mathcal{A})$ implies $\pi, n-i \models s_{n-i}$ holds for $0 \leq i \leq n$ by induction over the value of i.

Basis. $s_n \in Acc$ implies that $\epsilon \models s_n$, i.e., $\pi, n-0 \models s_{n-0}$.

Induction. The induction hypothesis is that $\pi, n-i \models s_{n-i}$ holds for $0 \leq i < n$. Note that $\delta(s_{n-(i+1)}, \pi[n-(i+1)]) = s_{n-i} = \mathsf{fp}(s_{n-(i+1)}, \pi[n-(i+1)])$. So the induction hypothesis is equivalent to $\pi, n-i \models \mathsf{fp}(s_{n-(i+1)}, \pi[n-(i+1)])$. By Lemma 14.2, we can get that $\pi, n-(i+1) \models s_{n-(i+1)}$.

Now we have proved that $\pi \in \mathcal{L}(\mathcal{A})$ implies $\pi, n-i \models s_{n-i}$ holds for $0 \leq i \leq n$. Then we can have $\pi \models \varphi$ (i.e., $\pi, n-n \models s_{n-n}$) holds when $i = n$. □

Theorem 14.6. *Let* φ *be an* LTL$_f$ *formula, the constructed DFA* $\mathcal{A}$ *can have, in the worst case,* $2^{2^{|\mathsf{cl}(\varphi)|}}$ *states.*

Proof. Having $\mathsf{cl}(\varphi)$, there is only $2^{\mathsf{cl}(\varphi)}$ possible models. Every state in $\text{Reach}(\varphi)/_{\sim_p}$ corresponds to a subset of $2^{\mathsf{cl}(\varphi)}$ possible models. That is to say, $\text{Reach}(\varphi)/_{\sim_p}$ can have, in the worst case, $2^{2^{|\mathsf{cl}(\varphi)|}}$ states that are not propositional equivalent. We thus show that $\mathcal{A}$ can have, in the worst case, $2^{2^{|\mathsf{cl}(\varphi)|}}$ states. □

14.3 LTL$_f$ Synthesis as AND-OR Graph Search

Recall that LTL$_f$ synthesis can be viewed as an *adversarial reachability* game on the DFA of the given formula. Interestingly, this game can actually be considered as an AND-OR graph, where the OR-nodes indicate the agent actions (quantified existentially), and the AND-nodes indicate the environment responses (quantified universally). In this case, the DFA construction approach described in the previous section allows us to solve LTL$_f$ synthesis via on-the-fly AND-OR graph search. Now, we present our approach of solving LTL$_f$ synthesis via on-the-fly AND-OR

graph search, and explain how to leverage KC techniques, Sentential Decision Diagrams (SDDs) (Darwiche, 2011) to significantly reduce the branching factor of the constructed graph.

14.3.1 Synthesis Algorithm

Given problem $(\varphi, \mathcal{X}, \mathcal{Y})$, our synthesis algorithm searches for a strategy by exploring the constructed AND-OR graph on the fly. This algorithm is basically a top-down traversal of the search space, proceeding forward from the initial, and excluding strategies that lead to loops. Since we apply the crucial step of propositional equivalence check whenever computing a new state, for simplicity, we omit the propositional equivalence symbol $\sim_p$ and denote every newly constructed DFA state by ψ, instead of $[\psi]_{\sim_p}$, e.g., the initial state is denoted by φ, instead of $[\varphi]_{\sim_p}$. Every DFA state is stored as an OR-node, each outgoing transition (or-arc) leads to an AND-node. Every or-arc is stored as an action-AndNode pair $(act, AndNd)$. Every outgoing transition (and-arc) of an AND-node is stored as a response-OrNode pair $(resp, n)$. A strategy is stored as a set of state-action pairs. If φ is unrealizable, we obtain strategy as an empty set. In order to avoid exploring the same state over and over, we assign a tag to its associated OR-node n after exploring it. More specifically, n is tagged as *success* if the corresponding DFA state ψ is accepting or there exists an act such that, regardless of what the environment $resp$ is, all corresponding followup OR-nodes are already tagged as *success*. In this case, we also add state-action pair (ψ, act) to strategy. If such act does not exist, n is tagged as *failure*. Moreover, we also put a *loop* tag on an OR-node n if a loop is detected on n, which is considered as temporary failure.

Algorithm 5 SDD-based Forward Synthesis

```
1: function Synthesis(φ) return strategy
2:     if IsAccepting(φ) then
3:         AddToStrategy(φ, true)
4:         return GetStrategy()
5:     InitialGraph(φ)
6:     n := GetGraphRoot()
7:     found := Search(n, ∅)
8:     if found then return GetStrategy()
9:     return EmptyStrategy() ▷ φ is unrealizable

10: function Search(n, path) return True/False
11:     if IsSuccessNode(n) then return True
12:     if IsFailureNode(n) then return False
13:     if InPath(n, path) then ▷ We found a loop
14:         TagLoop(n) return False
15:     ψ :=FormulaOfNode(n)
16:     if IsAccepting(ψ) then
17:         TagSuccessNode(n)
18:         AddToStrategy(ψ, true)
19:         return True
20:     Expand (n) ▷ Uses SDD to partition ψ wrt 𝒴 and 𝒳
21:     for (act, AndNd) ∈GetOrArcs(n) do
22:         for (resp, succ) ∈GetAndArcs(AndNd) do
23:             found :=Search(succ, [path|n])
24:             if ¬found then Break
25:         if found then
26:             TagSuccessNode(n)
27:             AddToStrategy(ψ, act)
28:             if IsTagLoop(n) then
29:                 BackProp(n)
30:             return True
31:     TagFailureNode(n)
32:     return False
```

Algorithm 6 Propagate Success Backwards

```
1: function BackProp(n)
2:     N := Enqueue(EpQueue,
3:     while !IsEmpty(N) do
4:         n_p := Dequeue(N)
5:         for (act, AndNd) ∈GetOrArcs(n_p) do
6:             if AllChildrenSuccess(AndNd) then
7:                 TagSuccessNode(n_p)
8:                 ψ :=FormulaOfNode(n_p)
9:                 AddToStrategy(ψ, act)
10:                Ns :=Enqueue (N, FailurePNs(n_p))
11:                Break
```

As shown in Algorithm 5, the Synthesis procedure takes a given LTL$_f$ formula φ as input ($\mathcal{X}, \mathcal{Y}$ are omitted for simplicity), and first checks whether the initial state φ is accepting. If this is the case, state-action pair (φ, true) is added to strategy and returned (Line 4). This is because the agent can do whatever it wants (i.e., assign any value to its variables $\mathcal{Y}$) after reaching an accepting state. Otherwise, we initialize the graph by creating an OR-node n out of φ, and start the main procedure Search. The Search procedure is a recursive routine, taking an OR-node n and the path leading to n as inputs, returning True (resp. False) indicating that a strategy is (resp. isn't) found by the current recursion. Hence, if the outmost Search returns True, a strategy consisting of all state-action pairs added until then is returned (Line 8). Otherwise, an empty strategy is returned.

Search processes an OR-node n by first checking whether n is tagged already, if so, it returns True for *success* tag, and False for *failure* tag. Then, if n exists on path thus leading to a loop, we put a *loop* tag on node n, and return False. Intuitively, when a loop is detected at node n, the procedure returns False, temporally considering n as a failure node. Note that we do not tag n as failure, since it is unknown here whether all the or-arcs of n are explored. Indeed, the returned False will be taken into account when tagging the ancestor nodes of n. Therefore, when later n is tagged as *success*, this information needs to be propagated back to the ancestor nodes of n.

Later on, the procedure continues by checking whether the associated DFA state ψ of n is accepting, if so, n is tagged as *success*, and (ψ, true) is added to the strategy. If none of these checks succeeds, n is expanded by Expand, which constructs all its or-arcs $(act, AndNd)$, and and-arcs $(resp, succ)$ of every *AndNd*. The crucial constraint is that all the agent actions *act*s of OR-node n should be disjoint and covering, the same with environment responses *resp* of every *AndNd*. Indeed, Expand is based on SDDs, see Section 14.3.2. As a side-effect, the Expand function stores the newly constructed nodes and arcs from n. We explore OR-node n, by iteratively processing the list of AND-nodes *AndNd* connected to n, until a strategy is found (Lines 21-32). In Line 23, we recursively call Search with the by n extended path. For every *AndNd*, once False is detected for searching some *succ*, we give up on the current *AndNd* and proceed with the next one (Line 24). If searching every *succ* of *AndNd* returns True (Line 25), n is tagged as *success*, and the corresponding state-action pair (ψ, act) is stored. Moreover, if n carries a *loop* tag, it is easy to see that n has been temporally considered as a *failure* node, and this information has been taken into account when tagging the ancestor nodes of n. Therefore, it is necessary to propagate this success information from n backwards to

the ancestor nodes of n (Lines 28-29). If no strategy is found after exploring n, we tag it as *failure*, and SEARCH returns False. It should be noted that, in a forward search on an AND-OR graph, it is critical to handle loops with the assistance of this backward propagation, by BACKPROP, as illustrated in (Scutellà, 1990).

As shown in Algorithm 6, BACKPROP is basically a bottom-up traversal of the subgraph rooted at n, that starts from the leaves, and propagates success backwards. In particular, only the nodes that are tagged as *failure* must be considered. This is because once a node n is tagged as *success*, it indicates that n is not affected by any temporary failure of its children. We start from the direct parents of n, and put them in a queue N. For every direct failure parent node n_p of n, n_p can be tagged as *success* only if there exists an or-arc $(act, AndNd)$, such that all the followup OR-nodes are already tagged as *success*. In this case, the corresponding state-action pair (ψ, act) is stored. Moreover, the success information of n_p should also be propagated, since n_p was tagged as *failure*, which could have affected the tag information of the direct parent nodes of n_p. Therefore, we add the *failure* nodes of them to N. The propagation continues until N gets empty. It is easy to see that the backward propagation does not change the forward nature of the SEARCH procedure, since the backward propagation has to be considered only as an instrument to correctly propagate the *success* whenever needed, i.e., in the presence of loops.

A major challenge arises, however, when looking into the branching factor of this AND-OR graph. Note that, in EXPAND, if we simply use $Y \in 2^{\mathcal{Y}}$ as *act* for every OR-node n, and $X \in 2^{\mathcal{X}}$ as *resp* for every *AndNd* connected to n, there can be far too many directions to explore, which leads to crucial performance limitation. Another challenge comes from the propositional equivalence check, which needs to be performed whenever computing a new state. We now explain how to use Sentential Decision Diagrams (SDDs) (Darwiche, 2011) to tackle both these challenges.

Algorithm 7 SDD-based ExpandGraph from An OrNode

```
1:  function EXPAND(n)
2:      ψ := FORMULAOFNODE(n)
3:      T := SDDREPRESENTATION(xnf(ψ))
4:      for child ∈ GETSDDCHILDREN(T) do
5:          act := GETSDDPRIME(child)
6:          AndNd := GETSDDSUB(child)
7:          ADDORARCS(n, act, AndNd)
8:          for child ∈ GETSDDCHILDREN(AndNd) do
9:              resp := GETSDDPRIME(child)
10:             sub := GETSDDSUB(child)
11:             succ := RMNEXT (sub)
12:             ADDANDARCS(AndNd, resp, succ)
```

14.3.2 SDD-based Expand

The crucial reason of adopting SDDs in the implementation of EXPAND (Algorithm 7), rather than other KC techniques, e.g., BDDs, is that, while maintaining canonicity to check propositional equivalence in constant time, SDDs can provide a disjoint, covering and compressed partition of a Boolean function, wrt a hierarchy

of $(\mathcal{Y}, \mathcal{X})$-partition. This allows us to easily partition the transition labels into disjoint agent moves and disjoint environment moves, compressed as much as possible, and so labeling transitions *symbolically* by propositional formulas. Let ψ be the associated DFA state of n, the input of EXPAND(n). The algorithm starts from computing $\mathsf{xnf}(\psi)$, which is equivalent to ψ, and intuitively, encodes all the possibilities of what happens *now*, expressed by $\mathcal{Y} \cup \mathcal{X}$ variables, and what happens *next* accordingly, expressed by variables $\mathcal{Z} = \bigcup_{\theta \in \mathsf{cl}(\varphi)} \{z_\alpha | \alpha \in \mathsf{pa}(\mathsf{xnf}(\theta)), \alpha \text{ not literal}\}$. Note that it is crucial to consider the closure of the original LTL$_f$ formula φ, instead of the current state ψ, as the propositional equivalence check between two states requires their SDDs to be defined over the same set of variables.

In Line 3, we represent $\mathsf{xnf}(\psi)$, considering it as a propositional formula over $\mathsf{pa}(\mathsf{xnf}(\psi))$, into an SDD $T := \bigvee_{i=1}^{n} [\mathsf{prime}_i(\mathcal{Y}) \wedge \mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})]$ such that all Ys leading to the same set of possible successors $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})$ (X is not decided yet) are clustered into a propositional formula $\mathsf{prime}_i(\mathcal{Y})$. $\mathsf{prime}_i(\mathcal{Y})$ and $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})$ are extracted as *act* and corresponding *AndNd*, respectively (Lines 5&6). Moreover, $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z}) = \bigvee_{j=1}^{m} [\mathsf{prime}_{i,j}(\mathcal{X}) \wedge \mathsf{sub}_{i,j}(\mathcal{Z})]$ is such that all Xs leading to the same successor are clustered into formula $\mathsf{prime}_j(\mathcal{X})$, and $\mathsf{sub}_{i,j}(\mathcal{Z})$ refers to the successor state of agent-env choices $(\mathsf{prime}_i(\mathcal{Y}), \mathsf{prime}_{i,j}(\mathcal{X}))$. They are extracted as *resp* and corresponding *succ*, respectively (Lines 9-11). Note that SDDs guarantee that all disjuncts generated are disjoint, covering and compressed, hence we can use SDDs to reduce the branching factor as much as possible. In particular, every successor state *succ* is obtained by stripping $\bigcirc$ and $\bullet$, introduced by XNF, through the *remove-next* function RMNEXT, defined below:

- RMNEXT$(\Diamond true) = tt$;
- RMNEXT$(\Box false) = ff$;
- RMNEXT$(\varphi_1 \wedge \varphi_2) =$ RMNEXT$(\varphi_1) \wedge$ RMNEXT(φ_2);
- RMNEXT$(\varphi_1 \vee \varphi_2) =$ RMNEXT$(\varphi_1) \vee$ RMNEXT(φ_2);
- RMNEXT$(\bigcirc \varphi) = \varphi \wedge \Diamond true$;
- RMNEXT$(\bullet \varphi) = \varphi \vee \Box false$.

Note that RMNEXT applies to neither $\mathcal{U}$-,$\mathcal{R}$- formulas, since they do not appear in XNF, nor literals (p, $\neg p$), since its input is a propositional formula over variables $\mathcal{Z}$ that does not contain literals.

Proposition 14.7. *Given an* LTL$_f$ *formula* φ *in* NNF, $\forall \sigma \in 2^{\mathcal{P}}$*:*

$$\mathsf{fp}(\varphi, \sigma) \equiv \mathit{RMNEXT}(\mathsf{xnf}(\varphi)^p|_\sigma)$$

where $\mathsf{xnf}(\varphi)^p|_\sigma$ *stands for evaluating* σ *on* $\mathsf{xnf}(\varphi)^p$*.*

Proof. We prove by a structural induction on φ.

- $\varphi = tt$. We have that $\mathsf{fp}(tt, \sigma) \equiv tt$ which is equivalent to RMNEXT$(\mathsf{xnf}(tt)^p|_\sigma)$, since $\mathsf{xnf}(tt) = tt$ by definition.
- $\varphi = ff$. We have that $\mathsf{fp}(ff, \sigma) \equiv ff$ which is equivalent to RMNEXT$(\mathsf{xnf}(ff)|_\sigma)$, since $\mathsf{xnf}(ff) = ff$ by definition.
- $\varphi = p$. We have that $\mathsf{fp}(p, \sigma) \equiv tt$ iff $p \in \sigma$ which is equivalent to RMNEXT$(\mathsf{xnf}(p)^p|_\sigma)$, since $\mathsf{xnf}(p) = p$ and $p|_\sigma \equiv true$ iff $p \in \sigma$.

- $\varphi = \neg p$. We have that $\mathsf{fp}(\neg p, \sigma) \equiv tt$ iff $p \notin \sigma$ which is equivalent to $\textsc{RmNext}(\mathsf{xnf}(\neg p)^p|_\sigma)$, since $\mathsf{xnf}(\neg p) = \neg p$ and $(\neg p)|_\sigma \equiv true$ iff $\neg p \in \sigma$.

- $\varphi = \varphi_1 \wedge \varphi_2$. Note that $\mathsf{fp}(\varphi_1 \wedge \varphi_2, \sigma) \equiv \mathsf{fp}(\varphi_1, \sigma) \wedge \mathsf{fp}(\varphi_2, \sigma)$. By induction hypotheses, $\mathsf{fp}(\varphi_1, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\varphi_1)^p|_\sigma)$, and $\mathsf{fp}(\varphi_2, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\varphi_2)^p|_\sigma)$. Thus $\mathsf{fp}(\varphi_1 \wedge \varphi_2, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\varphi_1 \wedge \varphi_2)^p|_\sigma)$.

- $\varphi = \varphi_1 \vee \varphi_2$. Note that $\mathsf{fp}(\varphi_1 \vee \varphi_2, \sigma) \equiv \mathsf{fp}(\varphi_1, \sigma) \vee \mathsf{fp}(\varphi_2, \sigma)$. By induction hypotheses, $\mathsf{fp}(\varphi_1, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\varphi_1)^p|_\sigma)$, and $\mathsf{fp}(\varphi_2, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\varphi_2)^p|_\sigma)$. Thus $\mathsf{fp}(\varphi_1 \vee \varphi_2, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\varphi_1 \vee \varphi_2)^p|_\sigma)$.

- $\varphi = \mathsf{O}\psi$. We have that $\mathsf{fp}(\mathsf{O}\psi, \sigma) = \psi \wedge \Diamond true$. By definition, we have that $\textsc{RmNext}(\mathsf{xnf}(\mathsf{O}\psi)^p|_\sigma) = \psi \wedge \Diamond true$, since $\mathsf{xnf}(\mathsf{O}\psi) = \mathsf{O}\psi$ and by definition of RmNext. Thus $\mathsf{fp}(\mathsf{O}\psi, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\mathsf{O}\psi)^p|_\sigma)$.

- $\varphi = \bullet\psi$. We have that $\mathsf{fp}(\bullet\psi, \sigma) = \psi \vee \Box false$. By definition, we have that $\textsc{RmNext}(\mathsf{xnf}(\bullet\psi)^p|_\sigma) = \psi \vee \Box false$, since $\mathsf{xnf}(\bullet\psi) = \bullet\psi$ and by definition of RmNext. Thus $\mathsf{fp}(\bullet\psi, \sigma) \equiv \textsc{RmNext}(\mathsf{xnf}(\bullet\psi)^p|_\sigma)$.

- $\varphi = \varphi_1 \,\mathcal{U}\, \varphi_2$. We have that $\mathsf{fp}(\varphi_1 \,\mathcal{U}\, \varphi_2, \sigma) = \mathsf{fp}(\varphi_2, \sigma) \vee (\mathsf{fp}(\varphi_1, \sigma) \wedge \mathsf{fp}(\mathsf{O}(\varphi_1 \,\mathcal{U}\, \varphi_2), \sigma))$, which is equivalent to $\mathsf{fp}(\varphi_2, \sigma) \vee (\mathsf{fp}(\varphi_1, \sigma) \wedge \varphi_1 \,\mathcal{U}\, \varphi_2 \wedge \Diamond true)$. By definition, we have that $\textsc{RmNext}(\mathsf{xnf}(\varphi_1 \,\mathcal{U}\, \varphi_2)^p|_\sigma) = \textsc{RmNext}(((\mathsf{xnf}(\varphi_2)^p \wedge \Diamond true) \vee (\mathsf{xnf}(\varphi_1)^p \wedge \mathsf{O}(\varphi_1 \,\mathcal{U}\, \varphi_2)))|_\sigma)$. By definition of RmNext, this is equivalent to $(\textsc{RmNext}(\mathsf{xnf}(\varphi_2)^p|_\sigma) \wedge \textsc{RmNext}(\mathsf{xnf}(\Diamond true)^p|_\sigma)) \vee (\textsc{RmNext}(\mathsf{xnf}(\varphi_1)^p)|_\sigma \wedge \textsc{RmNext}(\mathsf{xnf}(\mathsf{O}(\varphi_1 \,\mathcal{U}\, \varphi_2))^p|_\sigma))$, which is equivalent to $\textsc{RmNext}(\mathsf{xnf}(\varphi_2)^p|_\sigma) \vee (\textsc{RmNext}(\mathsf{xnf}(\varphi_1)^p)|_\sigma \wedge \varphi_1 \,\mathcal{U}\, \varphi_2 \wedge \Diamond true)$. The $\Diamond true$ in the first clause disappeared from the expression because $\textsc{RmNext}(\Diamond true) = tt$. The statement follows by structural induction and by syntactic equivalence.

- $\varphi = \varphi_1 \,\mathcal{R}\, \varphi_2$. We have that $\mathsf{fp}(\varphi_1 \,\mathcal{R}\, \varphi_2, \sigma) = \mathsf{fp}(\varphi_2, \sigma) \wedge (\mathsf{fp}(\varphi_1, \sigma) \vee \mathsf{fp}(\bullet(\varphi_1 \,\mathcal{R}\, \varphi_2), \sigma))$, which is equivalent to $\mathsf{fp}(\varphi_2, \sigma) \wedge (\mathsf{fp}(\varphi_1, \sigma) \vee \varphi_1 \,\mathcal{R}\, \varphi_2 \vee \Box false)$. By definition, we have that $\textsc{RmNext}(\mathsf{xnf}(\varphi_1 \,\mathcal{R}\, \varphi_2)^p|_\sigma) = \textsc{RmNext}(((\mathsf{xnf}(\varphi_2)^p \vee \Box false) \wedge (\mathsf{xnf}(\varphi_1)^p \vee \bullet(\varphi_1 \,\mathcal{R}\, \varphi_2)))|_\sigma)$. By definition of RmNext, this is equivalent to $\textsc{RmNext}(\mathsf{xnf}(\varphi_2)^p|_\sigma) \wedge (\textsc{RmNext}(\mathsf{xnf}(\varphi_1)^p)|_\sigma \vee \textsc{RmNext}(\bullet(\varphi_1 \,\mathcal{R}\, \varphi_2)))$, which is equivalent to $\textsc{RmNext}(\mathsf{xnf}(\varphi_2)^p|_\sigma) \wedge (\textsc{RmNext}(\mathsf{xnf}(\varphi_1)^p)|_\sigma \vee \varphi_1 \,\mathcal{R}\, \varphi_2 \vee \Box false)$. $\Box false$ disappeared from the expression because $\textsc{RmNext}(\Box false) = ff$. The statement follows by structural induction and by syntactic equivalence. □

Proposition 14.8. *Let φ be an* LTL$_f$ *formula, the constructed DFA $\mathcal{A}$ using SDD-based* Expand *can have, in the worst case, $2^{2^{|\mathsf{cl}(\varphi)|}}$ states.*

Proof. Note that in the SDD-based construction, we index every constructed state ψ (OR-node) by the SDD of $\mathsf{xnf}(\varphi)^p$. Moreover, the SDD variables consist of $\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}$, where $\mathcal{Z} = \bigcup_{\theta \in \mathsf{cl}(\varphi)} \{z_\alpha | \alpha \in \mathsf{pa}(\mathsf{xnf}(\theta)), \alpha \text{ not literal}\}$. Therefore, it is guaranteed that $\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}$ is the same size as $\mathsf{cl}(\varphi)$. Note that by the definition of transformation function $\mathsf{xnf}()$ and $\mathsf{pa}()$, $\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}$ can be obtained from $\mathsf{cl}(\varphi)$ by simply replacing $\theta \in \mathsf{cl}(\varphi)$ by $\mathsf{O}\theta$, if θ is an $\mathcal{U}$-formula, and $\theta \in \mathsf{cl}(\varphi)$ by $\bullet\theta$, if θ is a $\mathcal{R}$-formula. Again, every SDD-based constructed state corresponds to a subset of $2^{\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}}$ possible models. That is to say, there can be, in the worst case, $2^{2^{|\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}|}}$ states that are not propositional equivalent. We thus show that $\mathcal{A}$ constructed using SDD-based Expand can have, in the worst case, $2^{2^{|\mathsf{cl}(\varphi)|}}$ states. □

Lemma 14.9. *Algorithm 7 is correct, i.e., given an OR node n,* EXPAND *correctly expands the search graph.*

Proof. By definition, the expected outcome of the ***E***function is that constructs from n all its or-arcs $(act, AndNd)$, and and-arcs $(resp, succ)$ of every $AndNd$, and that all the agent actions acts of OR-node n must be disjoint, covering and compressed, the same with environment responses $resp$ of every $AndNd$ connected to n. Moreover, we have to prove that for every $\sigma \in 2^{\mathcal{Y}\cup\mathcal{X}}$, we have that there exists only one triple $(act_i, resp_{i,j}, succ_{i,j})$, i.e. a path from n to $succ$ that passes through the i-th or-arc $(act_i, AndNd_j)$ and the j-th and-arc of AND-node $AndNd_i$, such that $\sigma|_{\mathcal{Y}} \models act_i$, $\sigma|_{\mathcal{X}} \models resp_{i,j}$, and $succ_{i,j} = \delta([\psi]_{\sim p}, \sigma)$. In order to be correct, Algorithm 2 must satisfy all the above conditions.

Let ψ be the formula associated to n. By construction, the SDD representation of $\mathsf{xnf}(\psi)$ is $T = \bigvee_{i=1}^{n}[\mathsf{prime}_i(\mathcal{Y}) \wedge \mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})]$, and by Algorithm 2 we have $act_i = \mathsf{prime}_i(\mathcal{Y})$ and $AndNd_i = \mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})$. Again by construction, $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z}) = \bigvee_{j=1}^{m}[\mathsf{prime}_{i,j}(\mathcal{X}) \wedge \mathsf{sub}_{i,j}(\mathcal{Z})]$, and by Algorithm 2 we have $resp_{i,j} = \mathsf{prime}_{i,j}(\mathcal{X})$ and $succ_{i,j} = \textsc{RmNext}(\mathsf{sub}_{i,j}(\mathcal{Z}))$. By the properties of the SDD compilation, all the act_i, i.e. the set of primes prime_i of T, are disjoint and covering; that is, for all $Y \in 2^{\mathcal{Y}}$ there exists only one i such that $Y \models act_i$. Moreover, for the same arguments applied to $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})$, once i is fixed, for all $X \in 2^{\mathcal{X}}$, there is only one j such that $X \models resp_{i,j}$. This means that for every $\sigma \in 2^{\mathcal{Y}\cup\mathcal{X}}$ there exists only one pair $(act_i, resp_{i,j})$ such that $Y \models act_i$, where $Y = \sigma|_{\mathcal{Y}}$, and $X \models resp_{i,j}$, where $X = \sigma|_{\mathcal{X}}$. Furthermore, the guarantee that all the subs are compressed implies that there cannot be less $(act_i, AndNd_i$ pairs and, given i, there cannot be less $(resp_{i,j}, succ_{i,j})$ pairs.

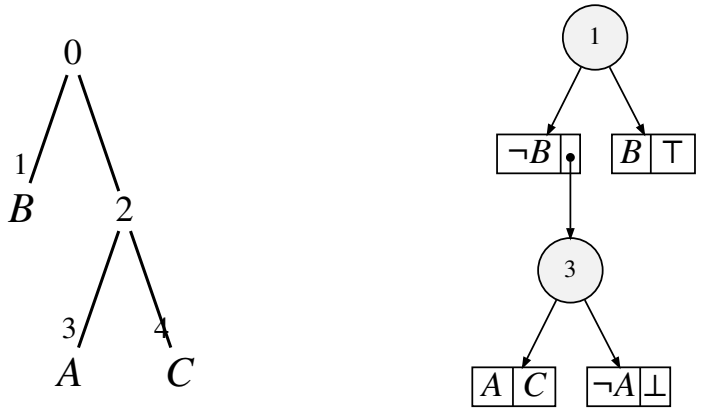
Given a pair $(act_i, resp_{i,j})$ of agent-env moves, let $succ_{i,j}$ be the associated successor OR-node extracted by the algorithm. To prove that $succ_{i,j}$ is indeed the right successor from node n after moves $(act_i, resp_{i,j})$, we note that by Lemma 14.7, we have that for every $\sigma \in 2^{\mathcal{P}}$,

$$\delta([\psi]_{\sim p}, \sigma) = \textsc{RmNext}(\mathsf{xnf}(\psi)^p\ |_\sigma) \tag{14.1}$$

From Equation 14.1, it follows that $succ_{i,j} = \textsc{RmNext}(\mathsf{sub}_{i,j}(\mathcal{Z}))$ is indeed the correct successor OR-node starting from node n following from agent move act_i and environment move $resp_{i,j}$. Together with Proposition 14.8, EXPAND also retains the doubly exponential blowup, we thus conclude that EXPAND is correct. □

Follows an example on how Lemma 14.9 can be applied:

Example 14.10. *Consider the* LTL$_f$ *formula $\varphi = a\,\mathcal{U}\,b$, where $\mathcal{Y} = \{b\}$ and $\mathcal{X} = \{a\}$. For the sake of simplicity, in this example we only consider one state variable, i.e., $\mathcal{Z} = \{\bigcirc(a\,\mathcal{U}\,b)\}$. The vtree is shown in Figure 14.1a. The disjoint outgoing transitions from state s_φ can be found by inspecting the SDD of $\mathsf{xnf}(\varphi)$, depicted in Figure 14.1b. In particular, the left child of the root (with index 1) describes the transition from the current OR-node, associated to formula φ, to the AND-node $AndNd_1$, represented by the SDD rooted at index 3, via agent move $\neg b$. From $AndNd_1$, the environment can do moves either a (left child) or $\neg a$ (right child). In the former case, the environment move a leads to corresponding* $\mathsf{sub}\ z_{\bigcirc a\,\mathcal{U}\,b}$*, that yields a successor OR-node associated to formula $\textsc{RmNext}(z_{\bigcirc a\,\mathcal{U}\,b}) = (a\,\mathcal{U}\,b) \wedge \Diamond true$. In the latter case, the environment move $\neg a$ leads to the successor OR-node associated to formula ff ($\bot$). which is a failure state, as expected. The right child of the original root (with index 1) describes the transition from the current OR-node, associated to formula φ, to the SDD denoting formula tt ($\top$) with agent move b (note, the*



(a) (b)

Figure 14.1. In Figure 14.1a, the vtree for $\varphi = a \,\mathcal{U}\, b$, with $\mathcal{Y} = \{B\}$, $\mathcal{X} = \{A\}$, $\mathcal{Z} = \{C\}$ and $A = a$, $B = b$, $C = \bigcirc(a \,\mathcal{U}\, b)$. For the sake of presentation, we only consider the state variable relevant in one step, and not all the subformulas of φ. In Figure 14.1b, the SDD of the formula $\mathsf{xnf}(\varphi) = b \vee (a \wedge \bigcirc(a \,\mathcal{U}\, b))$.

environment has no choice to prevent this). From the SDD denoting formula tt ($\top$), we extract the corresponding successor OR-node by means of the RmNext *function, giving* $tt = \textsc{RmNext}(tt)$.

Theorem 14.11. *Algorithm 5 terminates in at most double-exponential time, in the size of φ of problem $(\varphi, \mathcal{X}, \mathcal{Y})$.*

Proof. (Sketch) This is guaranteed by the fact that the number of recursive calls in Search is bounded by the worst-case doubly-exponential number of states in the constructed DFA via the SDD-based technique. Note that every recursive call first checks for *success*, *failure*, and loop (Lines 11-19). Then if the recursive call gets to Line 20, it will eventually tag the current node as *success* or *failure*. Therefore, every node is explored only once in a forward manner. Note that if a *success* node n was also tagged *loop*, BackProp is called before completing the current recursive call. BackProp is essentially a Breadth First Search (BFS) on the subgraph rooted at n. Since there can be at most linear number of $(n_1, (act, resp), n_2)$ edges in this subgraph, and every $(n_1, (act, resp), n_2)$ edge is visited only once during the BFS, BackProp terminates in linear time in the size of the subgraph rooted at n. Hence, we conclude that Algorithm 5 terminates in at most double-exponential time, in the size of φ of problem $(\varphi, \mathcal{X}, \mathcal{Y})$. □

Theorem 14.12. *Algorithm 5 is correct, i.e., it returns a non-empty strategy iff the given synthesis problem is realizable.*

Proof. (Sketch) We prove by showing the main recursive procedure Search is correct. If Search does not detect any loops on the graph, we can see that once an OR-node n is tagged, the tag is stored until the algorithm terminates. n is tagged as *success* if either n is detected as accepting, or there exists an agent *act*, following which, all the successors are also tagged as *success*, regardless of what the environment *resp* is, and the current recursion returns True. Note that, only in this case, the corresponding state-action pair is added to strategy. If neither condition happens, n is tagged as *failure*, and the current recursion returns False.

If SEARCH detects a loop on n in the graph, the presence of the loop leads to temporary failure. It could happen that a parent node n_p (also further ancestor nodes) of n is tagged as *failure* due to this temporary failure of n. Therefore, once n is tagged as *success*, the *success* tag should be propagated in the loops through BACKPROP. BACKPROP is correct, since the tag of a parent node n_p (also further ancestor nodes) changes from *failure* to *success* iff there exists an agent *act*, following which, all the successors are also tagged as *success*, regardless of what the environment *resp* is. Therefore, BACKPROP is able to eliminate the temporary failure caused by loops. Hence, if a *failure* tag stays until the algorithm terminates, this is a confirmed *failure* that is not affected by any temporary failure. As a result, Algorithm 5 terminates with the initial node tagged as *success* and returns a non-empty strategy iff the given synthesis problem is realizable. □

14.4 Related Work

Previous results on forward synthesis An attempt of applying forward LTL_f synthesis approach has been presented in (Xiao et al., 2021). That work presents an on-the-fly synthesis approach via conducting a so-called Transition-based Deterministic Finite Automata (TDFA) game, where the acceptance condition is defined on transitions, instead of states. Moreover, the game arena is constructed using SAT-based techniques. While that work successfully initiated research on on-the-fly LTL_f synthesis, its weaknesses motivated us to explore a different approach to forward-synthesis techniques. First, the transitions generated there are explicit, such that during search for every state, there can be an exponential number of options to explore. Second, due to the fact that the acceptance condition is defined on transitions, every generated transition has to be checked for acceptance. Furthermore, every state is represented explicitly as a Boolean formula, and thus the propositional equivalence check between two states leads to heavy cost, and there can be, in the worst case, doubly exponential number of states. Although both of the acceptance check and the propositional equivalence check can be done by utilizing sophisticated SAT-based techniques, the SAT solver is invoked in every single step, leading to unavoidable resource consumption.

POND planning Beyond FOND planning, another relevant work is planning in partially observable nondeterministic domains (POND), aka *contingent planning*, where the agent is only able to observe partly the current world state (Reif, 1984; Goldman and Boddy, 1996; Bonet and Geffner, 2000; Geffner and Bonet, 2013). In this case, the search space is no longer the set of the world states, but its powerset, a space of *belief-states*, which is in doubly-exponential size. In order to tackle this problem, several approaches have been employed, we look into three of them. The first one (Bertoli et al., 2006) leverages KC techniques to generate a symbolic, BDD-based representation of the planning domain, together with BDD representations of belief-states. In particular, when expanding the search space from a given node, one needs to evaluate the node on the domain BDD to obtain all of its possible outgoing edges, also represented as a BDD, which is then traversed to abstract the set of disjoint and covering outgoing edges. Note that in our approach, the search space is expanded by representing the XNF of the LTL_f formula relating to the current node as an SDD, which implicitly encodes the set of disjoint and covering outgoing edges. Representing the XNF formula in BDDs is also possible, since BDD is, in fact, a special case of SDD. The crucial benefit of using SDD is that, apart from representing propositional formulas, the disjoint, covering and

compressed partition (can be exponential size in the number of propositions) is naturally provided by SDD, without explicitly traversing its structure. It is indeed interesting to try different KC techniques in the future. The second approach to contingent planning, does not employ an explicit representation of belief-states during the computation, and instead represents a belief-state by an action-observation path leading to it (Hoffmann and Brafman, 2005). Therefore, the belief-state construction requires the knowledge of actions and the corresponding effects, which does not apply to the case of LTL$_f$ synthesis. Another approach to contingent planning applies an explicit disjunctive representation of belief-states (Thanh To, Pontelli, and Cao Son, 2009), since every belief-state is naturally a disjunction of conjuncted fluents. It should be noted that, LTL$_f$ formulas, even in NNF, allows arbitrary nesting of disjunctions and conjunctions. For that reason, one cannot directly apply their approach to LTL$_f$ synthesis.

14.5 Summary and Discussion

It presents a novel approach to solve LTL$_f$ synthesis. The proposed approach reduces the problem to AND-OR graph search, rather than first computing the DFA and then solving the DFA game (Chapter 13). This has the most important advantage that bypasses the full construction of the DFA of the LTL$_f$ formula, the major bottleneck for the synthesis solutions based on it. Instead, a forward approach only explores a subgraph of the full automaton. We observed that even an uninformed search, i.e., without heuristics, is able to drastically improve the synthesis capability in several cases. Crucially, it is important to be symbolic in the transition representation as otherwise the explicit representation of an alphabet over binary propositions does not scale with the number of propositions. That is why using techniques to reduce the branching factor at the minimum are necessary, and using SDDs is one of those, as we have shown.

There are different future research directions.

14.5.1 Informed Search

Instead of using a blind AND/OR search algorithm, one can use informed variants of AND/OR graph search algorithms, that make use of heuristics, like AO* (J. Nilsson, 1982; Pearl, 1984); in particular, the variants that also works for graphs with cycles (Mattmüller, 2013; Jiménez and Torras, 2000; Mahanti and Bagchi, 1985). This can lead to development of an entirely new research topic dealing with the study and discovery of good heuristics, either problem dependent or problem independent, that work well for the problem of LTL$_f$ synthesis.

14.5.2 Different Strategies to implement Expand

In the approach presented in this chapter, we use SDDs in the implementation of the EXPAND function (Algorithm 7). Of course, it is not the unique approach. One can still use BDDs, although their usage requires custom graph navigation in order to discover the disjoint and covering transitions. Instead of using knowledge compilation techniques, one can use SAT-based approaches to iteratively branch on formulas, hence avoiding to generate all the children nodes at once, leveraging the recent successes of the last two decades in the SAT solvers (Silva and Sakallah, 1996); or alternatively, design a custom DPLL-like search algorithm to explore disjoint and covering agent-env move pairs.

14.5.3 Extension to LDL$_f$, PPLTL, PPLDL

The technique is readily extendible to LDL$_f$ and to the computationally more tractable fragments of PPLTL and PPLDL. The computational advantage could be used for a straightforward symbolic state representation, where the bits are in one-to-one correspondence with the formula closure.

14.5.4 Other optimizations

Among other optimizations applicable, there are: on-the-fly minimization of the sub-DFA discovered during the search, and find smarter heuristics similar to look-ahead realizability and unrealizability checks.

Chapter 15

Cynthia

This chapter presents Cynthia, an implementation of the approach presented in Chapter 14.

The chapter is structured as follows:

- In Section 15.1, we briefly describe the software architecture and the internal working of Cynthia.
- In Section 15.2, we evaluate the implemented tool on LTL$_f$ synthesis benchmarks.
- Section 15.4 concludes the chapter, proposing future research directions.

15.1 Implementation

We implemented the forward synthesis problem presented in Section 14.3 in a tool called Cynthia in C++ [1]. Cynthia is able to take an LTL$_f$ synthesis problem $(\varphi, \mathcal{X}, \mathcal{Y})$ and constructs a strategy that realizes φ if one exists. The LTL$_f$ formula is parsed using Flex and Bison (Levine, 2009) with a custom grammar[2], and the syntactic tree is represented using n-ary trees. The partition of the variables is specified in a separate file. We make use of library SDD-2.0 (http://reasoning.cs.ucla.edu/sdd) to handle all SDD related operations.

Optimizations. Cynthia applies some optimizations to speed up the synthesis procedure. First, right before EXPAND an OR-node n, we perform the pre-processing techniques described in (Xiao et al., 2021). More specifically, we check: (i) there exists a one-step strategy that reaches accepting states from n, then n is tagged as success; or (ii) there does not exist an agent move that can avoid sink state (a non-accepting state only going back to itself) from n, then n is tagged as failure. Moreover, despite being a depth-first search, the SDD-based EXPAND(n), in fact, constructs all the connected *AndNd* of n, and followup OR-nodes *succ* at once, which allows us to conduct a "look-ahead" check. More specifically, this "look-ahead" check tries to tag constructed *succ* by the pre-processing techniques to speed up further search.

[1] Tool available at https://whitemech.github.io/cynthia.

[2] https://marcofavorito.me/tl-grammars/

15.2 Empirical Evaluation

Experimental Methodology. We evaluated the efficiency of Cynthia, by comparing against the following tools: Lisa (Bansal et al., 2020) and Lydia (De Giacomo and Favorito, 2021) are state-of-the-art backward LTL$_f$ synthesis approaches. Both tools compute the complete DFA first, and then solve an adversarial reachability game following the symbolic backward computation technique described in (Zhu, Tabajara, Li, et al., 2017). Ltlfsyn (Xiao et al., 2021) implements a SAT-based on-the-fly forward synthesis approach.
Experiment Setup. Experiments were run on a computer cluster, where each instance took exclusive access to a computing node with Intel-Xeon processor running at 2.6 GHz, with 8GB of memory and 300 seconds of time limit. The correctness of Cynthia was empirically verified by comparing the results with those from all baseline tools. No inconsistencies were encountered for all solved instances.

15.3 Empirical Evaluations

15.3.1 Benchmarks

We collected, in total, 1494 LTL$_f$ synthesis instances from literature, consisting of 3 benchmark families: 40 patterned instances from the *Patterns* benchmark family (Xiao et al., 2021), split into the GF-pattern and U-pattern datasets; 54 instances from the *Two-player-Games* benchmark family (Tabajara and Vardi, 2019a; Bansal et al., 2020), split into Single-Counter, Double-Counters and Nim datasets. Since the formulation there assumes that the environment acts first, the LTL$_f$ instances had to be modified slightly to adapt to our setting, where the agent acts first; 1400 randomly conjuncted instances taken from (Zhu, Tabajara, Li, et al., 2017; De Giacomo and Favorito, 2021).

Patterns. There are 20 unrealizable GF-pattern instances, and 20 realizable U-pattern instances, constructed in the following ways, respectively.

$$GF(n) = G(p_1) \wedge F(q_2) \wedge F(q_3) \wedge \ldots \wedge F(q_n)$$
$$U(n) = p_1 U(p_2 U(\ldots p_{n-1} U p_n))$$

More specifically, G stands for $\Box$ (Always), F stands for $\Diamond$ (Eventually), and U stands for $\mathcal{U}$ (Until). The variables in the formulas are roughly equally partitioned into $\mathcal{X}$ and $\mathcal{Y}$ at random. In particular, for GF-pattern instances, the first variable p_1 is always assigned as environment variable such that all generated instances are guaranteed to be unrealizable. Moreover, for U-pattern instances, the last variable p_n ($n \geq 2$) is always assigned as agent variable such that all generated instances are guaranteed to be realizable.

Random. This benchmark family has 1400 instances, from which there are 1000 instances from (Zhu, Tabajara, Li, et al., 2017), and 400 instances from (De Giacomo and Favorito, 2021). The instances in this benchmark family are constructed from basic cases taken from LTL synthesis datasets Lily (Jobstmann and Bloem, 2006) and Load balancer (Ehlers, 2010). Formally, a random conjunction formula $RC(L)$ has the form: $RC(L) = \bigwedge_{1 \leq i \leq L} P_i(v_1, v_2, ..., v_k)$, where L is the number of conjuncts, or the length of the formula, and P_i is a randomly selected basic case. Variables $v_1, v_2, \ldots, v_k$ are chosen randomly from a set of m candidate variables. Given L

and m (the size of the candidate variable set), we generate a formula $RC(L)$ in the following way:

1. Randomly select L basic cases;
2. For each case φ, substitute every variable v with a random new variable v' chosen from m atomic propositions. If v is an environment-variable in φ, then v' is also an environment-variable in $RC(L)$. The same applies to the agent-variables.

For the descriptions of *Single-Counter*, *Double-Counter*, *Nim* and *Random* see Section 7.3.3.

Plots. For better readability, we show the plots of experimental results in larger size here.

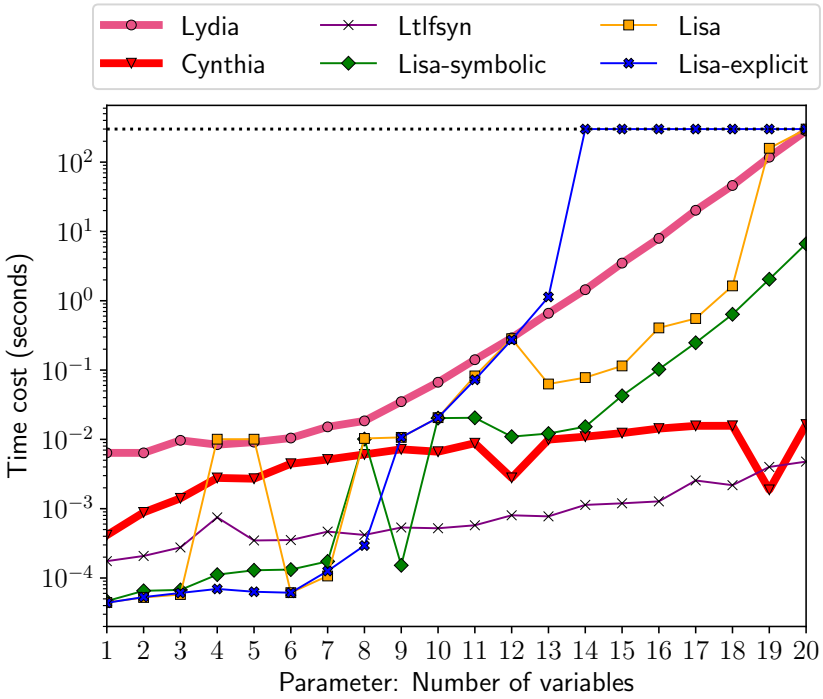


Figure 15.1. Results on GF-pattern.

Results. Figure 15.1 and Figure 15.2 show the running time of each tool on every instance of the GF-, and U-pattern, respectively. Across these instances, we observe that Cynthia is able to solve all instances with much less time comparing to backward approaches, represented by Lisa and Lydia.

Comparing to Ltlfsyn, Cynthia is able to achieve comparative performance on the GF-pattern instances, with time cost difference of <1 second (y-axis is in log scale), see Figure 15.1. On the U-pattern instances, Cynthia shows significantly better performance, see Figure 15.2. Detailed running times can be found in Table 15.1 for GF-pattern and in Table 15.2 for U-pattern instances.

On the *Two-player-Games* benchmarks, see Figure 15.3, we observe that Cynthia is able to dominate all other tools on the *Nim* instances. Yet, on both *Counter(s)* instances, backward approaches show better performance over all forward approaches,

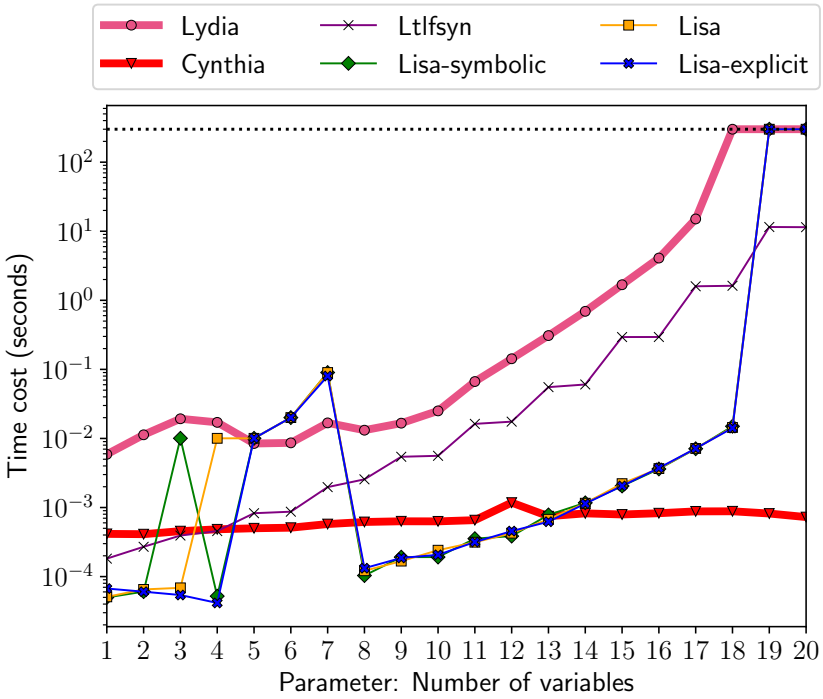


Figure 15.2. Results on U-pattern.

and Cynthia is almost on par with Ltlfsyn. Detailed running times can be found in Table 15.3 for *Nim* instances and in Table 15.4 and in Table 15.5 for single counters and double counters benchmark, respectively.

On the *Random* benchmarks, Cynthia, in general, performs better than Ltlfsyn, by solving more instances with less time, see Figure 15.4. Nevertheless, Cynthia cannot beat backward approaches.

Analysis. It is clear from the plots that Cynthia, in general, shows an overall better performance than Ltlfsyn, illustrating the efficiency and better scalability of our approach. In particular, there is a notable outperformance of Cynthia on the U-pattern instances, see Figure 15.2. The challenge in the U-pattern instances lies mostly in proving realizable, and can be achieved by just satisfying variables under control. Since every variable appears only once on the right side of the $\mathcal{U}$ operator, our approach is able to compress the branching labels as propositional formulas, such that highly reducing the branching factor and thus speeding up the search procedure.

When comparing Cynthia with backward approaches integrated tools, it should be noted that, in general, forward approaches perform well on the instances where the result can be obtained far before exploring the whole search space.

In our benchmarks, this is exactly what happens for *Nim* and *Pattern* instances, where Cynthia shows dominating performance over all tools, which demonstrates the promising efficiency of forward synthesis.

On the other hand, backward approaches perform better when it is necessary to explore the entire search space.

In the case of the *Counter(s)* instances, due to their specific structure, in order to obtain a strategy, the searching space to explore grows exponentially fast. In particular, the branching factor of AND-nodes, even after clustering, can remain

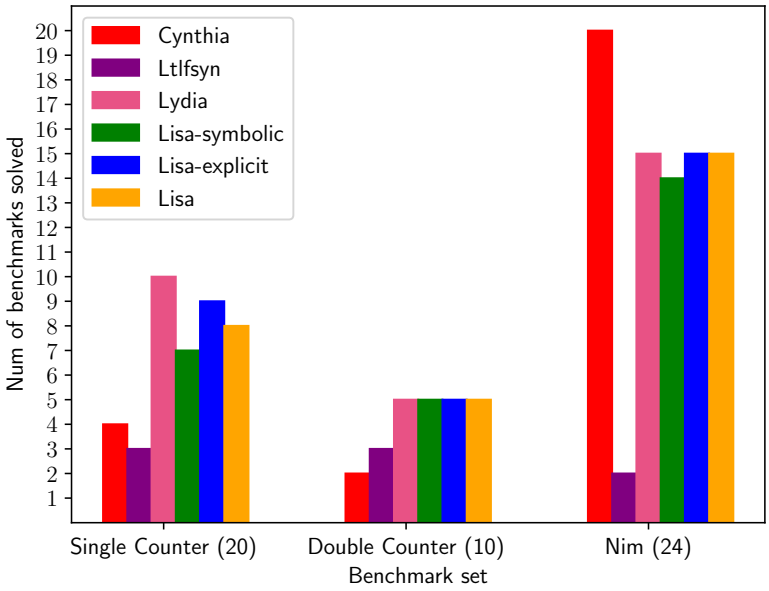


Figure 15.3. Results on *Two-player-Games*.

exponential in the number of environment variables, and so leaving no space to further reduce. Nevertheless, backward approaches can leverage powerful minimization to highly reduce the searching space such that achieving better performance, as also observed in (Tabajara and Vardi, 2019a).

For the *Random* instances, which are randomly conjuncted LTL$_f$ formulas, the advantage of possibly being lucky and finding a solution quickly without exploring the entire search space is overwhelmed by the fact that backward approaches integrated with composition techniques (Bansal et al., 2020; De Giacomo and Favorito, 2021) are able to first decompose the conjuncted formula into smaller pieces, obtain the minimized DFA of each conjunct and then compose them for final game solving. It might be possible that similar composition ideas could be leveraged to forward synthesis approaches as well, although further research is necessary in this direction.

15.4 Summary and Discussion

This chapter described Cynthia, an implementation of forward LTL$_f$ synthesis based on Sentential Decision Diagrams for minimizing the branching factor of AND/OR search graph induced by agent's and environment's moves. Experimental evaluation shows how it is superior than the other tool for forward LTL$_f$ synthesis, Ltlfsyn, which trivially evaluates all possible agent's and environment's moves, and it is competitive with the state-of-the-art tools Lydia and Lisa. As often happens in computer science, for hard problems, there is no "one size that fits all"; the best solution for LTL$_f$ synthesis would embrace the trade-off coming from forward search and backward search and chose the approach, or a mixture of them, that best fits the current problem.

Among future improvements of Cynthia, we would like to consider and address the following limitations:

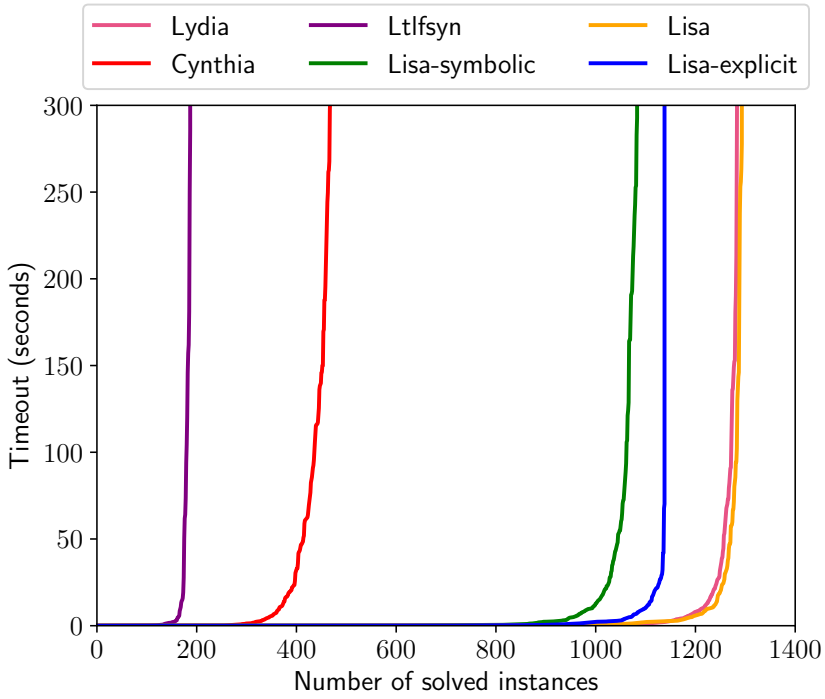


Figure 15.4. Results on *Random*.

- despite the SDD-based approach for computing all the node successors showed to be beneficial, there are cases in which the SDD compilation to compute the available moves and successors becomes the bottleneck of the procedure. Different strategies to solve this issue could be: (i) find better variable ordering; (ii) only consider the SDD state variables at the current level of decomposition, not all the state variables.

- From a design perspective, in the current implementation, the search procedure and the decomposition procedure are intertwined. This should be decoupled as much as possible, so to make the whole software architecture more modular and so easy to perform benchmarks across different versions of the same modules.

- Run more thorough benchmarks, considering different AND/OR search algorithms and benchmarks that highlight the trade-off between backward and forward search.

Table 15.1. Results on GF-pattern, time in milliseconds.

name	Cynthia	Ltlfsyn	Lydia	Lisa	Lisa-explicit	Lisa-symbolic
gfand01	0.41	0.18	6.39	0.04	0.04	0.05
gfand02	0.88	0.21	6.38	0.05	0.05	0.07
gfand03	1.41	0.28	9.69	0.06	0.06	0.07
gfand04	2.78	0.75	8.39	10.06	0.07	0.11
gfand05	2.71	0.35	9.13	10.06	0.06	0.13
gfand06	4.45	0.35	10.46	0.06	0.06	0.13
gfand07	5.12	0.47	15.16	0.11	0.13	0.17
gfand08	6.07	0.42	18.49	10.31	0.29	10.11
gfand09	7.18	0.54	34.99	10.68	10.70	0.15
gfand10	6.64	0.52	66.73	20.58	20.57	20.27
gfand11	8.80	0.58	141.08	82.34	72.39	20.49
gfand12	2.78	0.80	291.40	283.73	273.11	10.95
gfand13	9.92	0.78	660.43	62.95	1134.83	12.14
gfand14	10.95	1.13	1443.26	77.90	N/A	15.24
gfand15	12.31	1.20	3493.02	114.79	N/A	42.58
gfand16	14.31	1.28	7954.64	406.03	N/A	102.66
gfand17	15.67	2.55	20196.53	554.40	N/A	247.74
gfand18	15.71	2.18	46037.30	1639.00	N/A	637.70
gfand19	1.87	4.00	117824.24	158406.40	N/A	2043.10
gfand20	16.37	4.78	282788.90	N/A	N/A	6639.74

Table 15.2. Results on U-pattern, time in milliseconds.

name	Cynthia	Ltlfsyn	Lydia	Lisa	Lisa-explicit	Lisa-symbolic
uright01	0.42	0.18	5.90	0.05	0.07	0.05
uright02	0.41	0.27	11.29	0.07	0.06	0.06
uright03	0.46	0.39	19.27	0.07	0.05	10.06
uright04	0.48	0.46	17.10	10.04	0.04	0.05
uright05	0.50	0.83	8.45	10.04	10.04	10.04
uright06	0.51	0.87	8.63	20.04	20.04	20.04
uright07	0.58	1.98	16.80	90.06	80.05	90.04
uright08	0.62	2.56	13.17	0.12	0.13	0.10
uright09	0.63	5.43	16.64	0.17	0.19	0.19
uright10	0.63	5.61	25.07	0.24	0.21	0.19
uright11	0.66	16.26	66.65	0.31	0.32	0.35
uright12	1.18	17.51	142.48	0.42	0.46	0.39
uright13	0.75	55.34	308.67	0.68	0.62	0.78
uright14	0.82	60.44	692.79	1.15	1.14	1.17
uright15	0.79	293.86	1680.48	2.22	2.05	2.05
uright16	0.83	294.41	4090.18	3.66	3.74	3.61
uright17	0.88	1596.28	15064.68	7.16	7.16	7.09
uright18	0.88	1623.28	N/A	14.42	14.16	15.01
uright19	0.82	11520.30	N/A	N/A	N/A	N/A
uright20	0.73	11448.30	N/A	N/A	N/A	N/A

Table 15.3. Results on *Nim*, time in milliseconds.

name	Cynthia	Ltlfsyn	Lydia	Lisa	Lisa-explicit	Lisa-symbolic
nim_1_1	36.70	N/A	15.72	0.06	0.06	0.46
nim_1_2	57.50	N/A	24.89	10.16	10.16	10.64
nim_1_3	63.94	N/A	45.02	20.25	20.29	30.99
nim_1_4	55.30	N/A	182.58	60.33	40.42	51.31
nim_1_5	70.19	N/A	888.22	120.47	110.47	121.85
nim_1_6	89.09	N/A	4594.25	250.65	250.63	252.54
nim_1_7	126.13	N/A	16304.63	510.76	500.75	513.77
nim_1_8	173.29	N/A	50293.76	N/A	N/A	N/A
nim_2_1	110.86	63304.90	37.41	20.34	10.67	44.55
nim_2_2	525.35	N/A	378.10	150.86	131.11	237.44
nim_2_3	756.25	N/A	7236.98	821.78	811.78	962.45
nim_2_4	4696.17	N/A	66607.41	3383.80	3393.82	3728.46
nim_2_5	21709.21	N/A	N/A	N/A	N/A	N/A
nim_2_6	34876.06	N/A	N/A	N/A	N/A	N/A
nim_2_7	20660.01	N/A	N/A	N/A	N/A	N/A
nim_2_8	205167.27	N/A	N/A	N/A	N/A	N/A
nim_3_1	466.88	N/A	373.48	360.14	300.15	636.95
nim_3_2	10816.32	N/A	47814.58	5654.51	5666.39	7779.32
nim_3_3	N/A	N/A	N/A	N/A	N/A	N/A
nim_3_4	N/A	N/A	N/A	N/A	N/A	N/A
nim_4_1	7895.97	N/A	21373.38	7183.50	7213.57	13724.89
nim_4_2	N/A	N/A	N/A	N/A	N/A	N/A
nim_5_1	259369.96	N/A	N/A	162875.08	184813.46	N/A
nim_5_2	N/A	N/A	N/A	N/A	N/A	N/A

Table 15.4. Results on *Single-Counter*, time in milliseconds.

name	Cynthia	Ltlfsyn	Lydia	Lisa	Lisa-explicit	Lisa-symbolic
counter_01	46.93	12.70	7.99	10.09	10.09	10.22
counter_02	300.26	321.19	11.16	50.21	60.22	51.27
counter_03	714.14	19255.70	16.86	520.59	520.56	532.52
counter_04	3249.34	N/A	37.86	12.50	22.56	232.32
counter_05	N/A	N/A	84.93	95.76	41.15	2488.26
counter_06	N/A	N/A	264.65	589.94	144.03	25401.00
counter_07	N/A	N/A	887.72	5129.60	773.66	256830.00
counter_08	N/A	N/A	4042.69	201722.00	5024.00	N/A
counter_09	N/A	N/A	21789.13	N/A	44492.40	N/A
counter_10	N/A	N/A	102739.97	N/A	N/A	N/A
counter_11	N/A	N/A	N/A	N/A	N/A	N/A
counter_12	N/A	N/A	N/A	N/A	N/A	N/A
counter_13	N/A	N/A	N/A	N/A	N/A	N/A
counter_14	N/A	N/A	N/A	N/A	N/A	N/A
counter_15	N/A	N/A	N/A	N/A	N/A	N/A
counter_16	N/A	N/A	N/A	N/A	N/A	N/A
counter_17	N/A	N/A	N/A	N/A	N/A	N/A
counter_18	N/A	N/A	N/A	N/A	N/A	N/A
counter_19	N/A	N/A	N/A	N/A	N/A	N/A
counter_20	N/A	N/A	N/A	N/A	N/A	N/A

Table 15.5. Results on *Double-Counter*, time in milliseconds.

name	Cynthia	Ltlfsyn	Lydia	Lisa	Lisa-explicit	Lisa-symbolic
counters_01	61.29	9.61	14.68	80.13	80.13	60.45
counters_02	3496.84	463.86	41.02	10.95	20.77	24.91
counters_03	N/A	100060.00	219.92	132.98	81.72	216.96
counters_04	N/A	N/A	2353.53	1263.96	446.48	2590.23
counters_05	N/A	N/A	41203.34	23491.22	2532.84	70869.70
counters_06	N/A	N/A	N/A	N/A	N/A	N/A
counters_07	N/A	N/A	N/A	N/A	N/A	N/A
counters_08	N/A	N/A	N/A	N/A	N/A	N/A
counters_09	N/A	N/A	N/A	N/A	N/A	N/A
counters_10	N/A	N/A	N/A	N/A	N/A	N/A

Chapter 16

Conclusions

As we have seen in Chapter 1, there is a lot of interest in applying temporal logics on finite traces to many relevant AI problems. In line with the recent interest in applying temporal logic on finite traces to AI problems, this thesis is about the advancement of the techniques based on the temporal logics $\text{LTL}_f/\text{LDL}_f$, as well as the formalization of novel interesting AI problems, and solutions based on applications of the aforementioned techniques. This thesis addressed several important topics in this field.

Regarding the construction of the DFA from an $\text{LTL}_f/\text{LDL}_f$ formula, we proposed a novel translation approach, which we call *compositional* due to its way of decomposing formulas up to base cases and handle them in a bottom-up fashion. Despite its computational complexity is non-elementary, which is worse than the classical translation that relies on the encoding in an alternating finite automaton followed by a determinization step, we showed that in practice this is not the case, thanks also to the opportunity of applying aggressive minimization after the computation of every partial result. Then, we formalized a more concrete variant of the approach that relies on automata representations and operations that are semi-symbolic, i.e. still explicit in the state space representation, but symbolic in the alphabet representation. This was a fundamental constrain to guarantee the scalability to big formulas, especially with respect to the number of variables. Finally, we implemented the semi-symbolic compositional approach in the tool *Lydia*, and showed how it is competitive with other state-of-the-art tools, both for DFA construction and for compositional LTL_f synthesis.

This contribution can make a big impact in the AI community that deals with temporal logics such as $\text{LTL}_f/\text{LDL}_f$, due to its foundational nature. Any tecnhique or algorithm that relies on the $\text{LTL}_f/\text{LDL}_f$-DFA connection can promptly benefit from our work. Moreover, it opens several future directions of research, e.g. by applying the compositional approach for other translations like LTL_f, PPLTL and PPLDL, and developments of optimizations for more efficient algorithms.

Then, we contributed to the problem of reinforcement learning with temporal logic specifications. In particular, we introduced the novel concept of Restraining Bolts, in the case where the agent is a reinforcement learning agent and the restraining specifications are $\text{LTL}_f/\text{LDL}_f$ formulas. We developed a technique to reduce the problem to classical reinforcement learning with an extended agent's state space, and formally proved that the correlation between the representations used for the features and the fluents of the authority does not need to be formalized in order for the agent to learn to satisfy the specification. We also contributed to how to engineer such temporal reward specifications by means of transducers, which reduce

the overhead at minimum, and allow to have a finer-grained control to the conditions that allow an agent to be rewarded, using the known conditions in the runtime monitoring literature. Among other applications, we developed an imitation learning method for heterogeneous agents, and a way to integrate a planning model with a learning agent in a domain-independent way.

Our contributions have the potential to open many new applications and theoretical developments to the employment of $\text{LTL}_f/\text{LDL}_f$ for reinforcement learning. Indeed, these high-level languages like $\text{LTL}_f/\text{LDL}_f$ might be useful as intermediate formats that are somewhat easy for people to write, but also somewhat easy for machines to interpret, in contrast with the "low-level programming" of transition-based rewards. The specification of rewards by means of temporal logics has been advocated by top scientists in reinforcement learning, like Michael Littman (Littman, 2015b; Littman, 2022), witnessing how it is of paramount importance finding good formalisms and approaches to synthetize a reward function in order to instruct the agent to learn to perform the desired task, whilst making sure the agent perofrms the task as intended. The recent literature in RL started to be sensible to this problem, as happened for an observed phenomenon like *reward hacking* (Clark and Amodei, 2016), where the agent exploits a misspecified reward function to gain more rewards in an unintended way. More generally, our contribution can be very useful to the recent interest in AI safety for reinforcement learning agents (Amodei et al., 2016; Garcia and Fernandez, 2015; Leike et al., 2017; Ray, Achiam, and Amodei, 2019), and some work already started using temporal logics for safe reinforcement learning, e.g. *shielding* (Alshiekh et al., 2018). The use of temporal logics in RL can be also important for improving sample efficiency and to inject prior knowledge to the task of the agent, by providing shaping rewards in a systematic and sound way (Grzes, 2010).

Finally, besides compositional LTL_f synthesis, this thesis made another contribution contributes to the problem of LTL_f synthesis by studying a very promising approach based on forward AND-OR graph search. Indeed, we reduced LTL_f synthesis to an instance of AND/OR graph search with cycles, and showed how to rely on knowledge compilation technique to make the search procedure over the implicit graph more practical, by clustering together equivalent agent's moves and environment's moves, and by allowing fast constant-time checking of state (syntactic) equivalence. We also provided an implementation, *Cynthia*, and experimentally proved how a forward approach can be beneficial to this problem for many instances.

As we said, the synthesis problem in finite settings is similar to reactive synthesis, except that the interaction of the agent with the environment is guaranteed to terminate. This fact makes finite-trace formalisms better suitable for modeling AI problems, where usually the task is guaranteed to terminate. However, despite having an elegant theory, much work has to be done to make such techniques of practical use. In fact, development of good algorithmic solutions is fundamental for coping with the intractability of the problem, whose complexity is 2EXPTIME-complete. The deeper connection between LTL_f synthesis and forward search opens countless developments, especially regarding the analogies with FOND planning. As we have shown, for several instances, it is actually easier to look for a winning strategy starting from the initial state of the system and move forward rather than starting from the winning states and going backward by using least-fixpoint computation. This observation, combined with a more efficient compilation of the search graph, and a development of suitable heuristics for this specific problem, possibly exploiting the structure of the formula, and so by using informed search algorithms, suggests many avenues for future research, with the potential of being very impactful.

Bibliography

Aalst, Wil M. P. van der. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

Abadi, Eden and Ronen I. Brafman. "Learning and Solving Regular Decision Processes". In: *IJCAI*. ijcai.org, 2020, pp. 1948–1954.

Abbeel, Pieter, Morgan Quigley, and Andrew Y. Ng. "Using inaccurate models in reinforcement learning". In: *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*. 2006, pp. 1–8. DOI: 10.1145/1143844.1143845. URL: https://doi.org/10.1145/1143844.1143845.

Abiteboul, Serge, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

Achiam, Joshua et al. "Constrained Policy Optimization". In: *ICML*. 2017, pp. 22–31.

Aksaray, Derya et al. "Q-Learning for robust satisfaction of signal temporal logic specifications". In: *CDC*. IEEE, 2016, pp. 6565–6570.

Almagor, Shaull, Udi Boker, and Orna Kupferman. "Formally Reasoning About Quality". In: *J. ACM* 63.3 (2016), 24:1–24:56.

Alshiekh, Mohammed et al. "Safe Reinforcement Learning via Shielding". In: *AAAI*. AAAI Press, 2018, pp. 2669–2678.

Althoff, Christoph Schulte, Wolfgang Thomas, and Nico Wallmeier. "Observations on Determinization of Büchi Automata". In: *CIAA*. Vol. 3845. Lecture Notes in Computer Science. Springer, 2005, pp. 262–272.

Alur, Rajeev, Loris D'Antoni, and Mukund Raghothaman. "DReX: A Declarative Language for Efficiently Evaluating Regular String Transformations". In: *POPL*. ACM, 2015, pp. 125–137.

Aminof, Benjamin, Giuseppe De Giacomo, Alessio Lomuscio, et al. "Synthesizing Best-effort Strategies under Multiple Environment Specifications". In: *KR*. 2021, pp. 42–51.

— "Synthesizing strategies under expected and exceptional environment behaviors". In: *IJCAI*. ijcai.org, 2020, pp. 1674–1680.

Aminof, Benjamin, Giuseppe De Giacomo, Aniello Murano, et al. "Planning under LTL Environment Specifications". In: *ICAPS*. AAAI Press, 2019, pp. 31–39.

Aminof, Benjamin, Giuseppe De Giacomo, and Sasha Rubin. "Best-Effort Synthesis: Doing Your Best Is Not Harder Than Giving Up". In: *IJCAI*. ijcai.org, 2021, pp. 1766–1772.

Amodei, Dario et al. "Concrete Problems in AI Safety". In: *CoRR* abs/1606.06565 (2016).

Andreas, Jacob, Dan Klein, and Sergey Levine. "Modular Multitask Reinforcement Learning with Policy Sketches". In: *ICML*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 166–175.

Angluin, Dana. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106.

Angluin, Dana, Sarah Eisenstat, and Dana Fisman. "Learning Regular Languages via Alternating Automata". In: *IJCAI*. 2015.

Arora, Saurabh and Prashant Doshi. *A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress*. 2018. arXiv: 1806.06877 [cs.LG].

Bacchus, Fahiem, Craig Boutilier, and Adam J. Grove. "Rewarding Behaviors". In: *AAAI/IAAI, Vol. 2*. AAAI Press / The MIT Press, 1996, pp. 1160–1167.

Bacchus, Fahiem and Froduald Kabanza. "Planning for Temporally Extended Goals". In: *AAAI/IAAI, Vol. 2*. AAAI Press / The MIT Press, 1996, pp. 1215–1222.

— "Planning for temporally extended goals". In: *Annals of Mathematics and Artificial Intelligence* 22.1 (1998), pp. 5–27.

— "Using temporal logics to express search control knowledge for planning". In: *Artif. Intell.* 116.1-2 (2000), pp. 123–191.

Bahar, R. Iris et al. "Algebraic Decision Diagrams and Their Applications". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 171–206.

Baier, C. and JP. Katoen. *Principles of model checking*. 2008.

Baier, Jorge A., Christian Fritz, Meghyn Bienvenu, et al. "Beyond Classical Planning: Procedural Control Knowledge and Preferences in State-of-the-art Planners". In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. AAAI'08. Chicago, Illinois: AAAI Press, 2008, pp. 1509–1512. ISBN: 978-1-57735-368-3. URL: http://dl.acm.org/citation.cfm?id=1620270.1620321.

Baier, Jorge A., Christian Fritz, and Sheila A. McIlraith. "Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners". In: *ICAPS*. AAAI, 2007, pp. 26–33.

Baier, Jorge A. and Sheila A. McIlraith. "Planning with First-Order Temporally Extended Goals using Heuristic Search". In: *Proc. of AAAI*. 2006, pp. 788–795.

Bansal, Suguman et al. "Hybrid Compositional Reasoning for Reactive Synthesis from Finite-Horizon Specifications". In: *AAAI*. AAAI Press, 2020, pp. 9766–9774.

Barto, Andrew G., Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Trans. Syst. Man Cybern.* 13 (Sept. 1983), pp. 834–846.

Basin, David and Nils Klarlund. "Automata based symbolic reasoning in hardware verification". In: *Formal Methods In System Design* 13.3 (1998), pp. 253–286.

— "Beyond the finite in automatic hardware verification". In: *Technical Report* (1996).

Basin, David A. and Nils Klarlund. "Hardware Verification using Monadic Second-Order Logic". In: *CAV*. Vol. 939. Lecture Notes in Computer Science. Springer, 1995, pp. 31–41.

Bauer, Andreas, Martin Leucker, and Christian Schallhart. "Comparing LTL Semantics for Runtime Verification". In: *J. Log. Comput.* 20.3 (2010), pp. 651–674.

Bellman, Richard. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957. URL: http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false.

Bertoli, P. et al. "Strong planning under partial observability". In: *Artif. Intell.* 170.4-5 (2006).

Bienvenu, Meghyn, Christian Fritz, and Sheila A. McIlraith. "Planning with Qualitative Temporal Preferences". In: *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*. Ed. by Patrick Doherty, John Mylopoulos, and Christopher A. Welty. AAAI Press, 2006.

Bienvenu, Meghyn, Christian Fritz, and Sheila A. McIlraith. "Specifying and computing preferred plans". In: *Artif. Intell.* 175.7-8 (2011), pp. 1308–1345.

Bloem, Roderick et al. "Symbolic Implementation of Alternating Automata". In: *Int. J. Found. Comput. Sci.* 18.4 (2007), pp. 727–743.

Bogdanov, Andrej and Luca Trevisan. "Average-Case Complexity". In: *Found. Trends Theor. Comput. Sci.* 2.1 (2006).

Bonet, B. and H. Geffner. "Planning with Incomplete Information as Heuristic Search in Belief Space". In: *AIPS*. 2000.

Bouajjani, Ahmed, Peter Habermehl, and Tomás Vojnar. "Abstract Regular Model Checking". In: *CAV*. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 372–386.

Bouton, C. L. "Nim, A Game with a Complete Mathematical Theory". In: *Annals of Mathematics* 3 (1901).

Brachman, Ronald J. and Hector J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004. ISBN: ['9781558609327']. DOI: 10.1016/b978-1-55860-932-7.x5083-3. URL: http://dx.doi.org/10.1016/b978-1-55860-932-7.x5083-3.

Brafman, R. I., G. De Giacomo, and F. Patrizi. "LTL$_f$/LDL$_f$ Non-Markovian Rewards". In: *AAAI*. 2018.

Brafman, Ronen I and Giuseppe De Giacomo. "Regular Decision Processes: A Model for Non-Markovian Domains." In: *IJCAI*. 2019, pp. 5516–5522.

— "Planning for LTL$_f$ /LDL$_f$ Goals in Non-Markovian Fully Observable Nondeterministic Domains". In: *IJCAI*. Ed. by Sarit Kraus. 2019.

Brafman, Ronen I., Giuseppe De Giacomo, and Fabio Patrizi. "Specifying Non-Markovian Rewards in MDPs Using LDL on Finite Traces (Preliminary Version)". In: *CoRR* abs/1706.08100 (2017).

Brafman, Ronen I. and Moshe Tennenholtz. "R-MAX - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning". In: *J. Mach. Learn. Res.* 3 (2002), pp. 213–231. URL: http://jmlr.org/papers/v3/brafman02a.html.

Brooks, Rodney A. "Intelligence without Representation". In: *Artif. Intell.* 47.1-3 (1991), pp. 139–159.

Bryant, Randal E. "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams". In: *ACM Comput. Surv.* 24.3 (1992), pp. 293–318.

Brzozowski, Janusz A. and Ernst L. Leiss. "On Equations for Regular Languages, Finite Automata, and Sequential Networks". In: *Theor. Comput. Sci.* 10 (1980), pp. 19–35.

Buchi, J. Richard and Lawrence H. Landweber. "Solving Sequential Conditions by Finite-State Strategies". In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311. ISSN: 00029947. URL: http://www.jstor.org/stable/1994916 (visited on 05/29/2022).

Büchi, J Richard. "Weak second-order arithmetic and finite automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960).

Büchi, Richard J. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92. DOI: 10.1002/malq.19600060105. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19600060105.

Burch, Jerry R. et al. "Symbolic Model Checking: 1020 States and Beyond". In: *Inf. Comput.* 98.2 (1992), pp. 142–170.

Calvanese, Diego, Giuseppe De Giacomo, and Moshe Y. Vardi. "Reasoning about Actions and Planning in LTL Action Theories". In: *KR*. 2002, pp. 593–602.

Camacho, Alberto, Jorge A. Baier, et al. “Finite LTL Synthesis as Planning”. In: *ICAPS*. 2018, pp. 29–38.
Camacho, Alberto, Meghyn Bienvenu, and Sheila A. McIlraith. “Finite LTL Synthesis with Environment Assumptions and Quality Measures”. In: *KR*. AAAI Press, 2018, pp. 454–463.
Camacho, Alberto, Oscar Chen, et al. “Decision-Making with Non-Markovian Rewards: From LTL to automata-based reward shaping”. In: *RLDM*. 2017, pp. 279–283.
— “Non-Markovian Rewards Expressed in LTL: Guiding Search Via Reward Shaping”. In: *SOC*. 2017, pp. 159–160.
Camacho, Alberto, Rodrigo Toro Icarte, et al. “LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning”. In: *IJCAI*. ijcai.org, 2019, pp. 6065–6073.
Camacho, Alberto and Sheila A McIlraith. “Strong Fully Observable Non-Deterministic Planning with LTL and LTLf Goals.” In: *IJCAI*. 2019, pp. 5523–5531.
— “Learning Interpretable Models Expressed in Linear Temporal Logic”. In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*. 2019, pp. 621–630. URL: https://aaai.org/ojs/index.php/ICAPS/article/view/3529.
Camacho, Alberto, Eleni Triantafillou, et al. “Non-Deterministic Planning with Temporally Extended Goals: LTL over Finite and Infinite Traces”. In: *AAAI*. 2017.
Chandra, Ashok K., Dexter Kozen, and Larry J. Stockmeyer. “Alternation”. In: *J. ACM* 28.1 (1981), pp. 114–133.
Church, Alonzo. “A note on the Entscheidungsproblem”. In: *The journal of symbolic logic* 1.1 (1936), pp. 40–41.
— “Application of recursive arithmetic to the problem of circuit synthesis”. In: *Journal of Symbolic Logic* 28.4 (1963).
— “Logic, arithmetic, and automata”. In: *Journal of Symbolic Logic* 29.4 (1964).
Cimatti, A., M. Pistore, et al. “Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking.” In: *Artificial Intelligence* 1–2.147 (2003).
Cimatti, A., M. Roveri, and P. Traverso. “Strong Planning in Non-Deterministic Domains Via Model Checking”. In: *AIPS*. 1998.
Cimatti, Alessandro, Fausto Giunchiglia, et al. “Planning via Model Checking: A Decision Procedure for *AR*”. In: *ECP*. Vol. 1348. Lecture Notes in Computer Science. Springer, 1997, pp. 130–142.
Ciolek, Daniel Alfredo et al. “Multi-Tier Automated Planning for Adaptive Behavior”. In: *ICAPS*. AAAI Press, 2020, pp. 66–74.
Clark, Jack and Dario Amodei. “Faulty reward functions in the wild”. In: *Internet: https://blog.openai.com/clark2016faultys* (2016).
Clarke, Edmund M., E. Allen Emerson, and A. Prasad Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263.
Clarke Jr., Edmund M., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.
D’Antoni, Loris and Margus Veanes. “Automata modulo theories”. In: *Commun. ACM* 64.5 (2021), pp. 86–95. DOI: 10.1145/3419404. URL: https://doi.org/10.1145/3419404.
— “The Power of Symbolic Automata and Transducers”. In: *CAV (1)*. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 47–67.
Darwiche, A. “SDD: A New Canonical Representation of Propositional Knowledge Bases”. In: *IJCAI*. 2011.

Davis, Martin. “Influences of mathematical logic on computer science”. In: *A half-century survey on The Universal Turing Machine*. 1988, pp. 315–326.

— *The universal computer: The road from Leibniz to Turing*. AK Peters/CRC Press, 2018.

De Giacomo, G. and M. Y. Vardi. “Automata-Theoretic Approach to Planning for Temporally Extended Goals”. In: *Recent Advances in AI Planning, 5th European Conference on Planning, ECP’99, Durham, UK, September 8-10, 1999, Proceedings*. Vol. 1809. Lecture Notes in Computer Science. Springer, 1999.

De Giacomo, Giuseppe, Riccardo De Masellis, and Marco Montali. “Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI’14. Québec City, Québec, Canada: AAAI Press, 2014, pp. 1027–1033. URL: http://dl.acm.org/citation.cfm?id=2893873.2894033.

De Giacomo, Giuseppe, Riccardo De Masellis, et al. “Monitoring Constraints and Metaconstraints with Temporal Logics on Finite Traces”. In: *CoRR* abs/2004.01859 (2020).

De Giacomo, Giuseppe, Antonio Di Stasio, et al. “Pure-Past Linear Temporal and Dynamic Logic on Finite Traces”. In: *IJCAI*. 2020.

De Giacomo, Giuseppe and Marco Favorito. “Compositional Approach to Translate LTLf/LDLf into Deterministic Finite Automata”. In: *ICAPS*. AAAI Press, 2021, pp. 122–130.

De Giacomo, Giuseppe, Marco Favorito, and Francesco Fuggitti. “Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic: A Polynomial Reduction to Standard Planning”. In: *arXiv preprint arXiv:2204.09960* (2022).

De Giacomo, Giuseppe, Marco Favorito, Luca Iocchi, and Fabio Patrizi. “Imitation Learning over Heterogeneous Agents with Restraining Bolts”. In: *ICAPS*. AAAI Press, 2020, pp. 517–521.

De Giacomo, Giuseppe, Marco Favorito, Luca Iocchi, Fabio Patrizi, and Alessandro Ronca. “Temporal Logic Monitoring Rewards via Transducers”. In: *KR*. 2020, pp. 860–870.

De Giacomo, Giuseppe, Marco Favorito, Luca Iocchi, et al. “Domain-independent reward machines for modular integration of planning and learning”. In: (2021).

De Giacomo, Giuseppe, Marco Favorito, Li Jianwen, et al. “LTLf Synthesis as AND-OR Graph Search”. In: *IJCAI (to appear)*. 2022.

De Giacomo, Giuseppe, Luca Iocchi, et al. “Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications”. In: *ICAPS*. AAAI Press, 2019, pp. 128–136.

— “Reinforcement Learning for LTLf/LDLf Goals”. In: *CoRR* arXiv:1807.06333 (2018).

De Giacomo, Giuseppe, Fabrizio Maria Maggi, et al. “On the Disruptive Effectiveness of Automated Planning for LTL*f*-Based Trace Alignment”. In: *AAAI*. AAAI Press, 2017, pp. 3555–3561.

De Giacomo, Giuseppe, Riccardo De Masellis, Marco Grasso, et al. “Monitoring Business Metaconstraints Based on LTL and LDL for Finite Traces”. In: *BPM*. Vol. 8659. Lecture Notes in Computer Science. Springer, 2014, pp. 1–17.

De Giacomo, Giuseppe, Riccardo De Masellis, and Marco Montali. “Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness”. In: *AAAI*. AAAI Press, 2014, pp. 1027–1033.

De Giacomo, Giuseppe and Sasha Rubin. “Automata-Theoretic Foundations of FOND Planning for LTLf and LDLf Goals”. In: *IJCAI*. ijcai.org, 2018, pp. 4729–4735.

De Giacomo, Giuseppe, Antonio Di Stasio, Giuseppe Perelli, et al. "Synthesis with Mandatory Stop Actions". In: *KR*. 2021, pp. 237–246.

De Giacomo, Giuseppe, Antonio Di Stasio, Lucas M. Tabajara, et al. "Finite-Trace and Generalized-Reactivity Specifications in Temporal Synthesis". In: *IJCAI*. ijcai.org, 2021, pp. 1852–1858.

De Giacomo, Giuseppe, Antonio Di Stasio, Moshe Y. Vardi, et al. "Two-Stage Technique for LTLf Synthesis Under LTL Assumptions". In: *KR*. 2020, pp. 304–314.

De Giacomo, Giuseppe and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *IJCAI*. IJCAI/AAAI, 2013, pp. 854–860.

— "LTLf and LDLf Synthesis under Partial Observability". In: *IJCAI*. IJCAI/AAAI Press, 2016, pp. 1044–1050.

— "Synthesis for LTL and LDL on Finite Traces". In: *IJCAI*. AAAI Press, 2015, pp. 1558–1564.

Devlin, Sam and Daniel Kudenko. "Dynamic Potential-based Reward Shaping". In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '12. Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 433–440. ISBN: 0-9817381-1-7, 978-0-9817381-1-6. URL: http://dl.acm.org/citation.cfm?id=2343576.2343638.

Dietterich, Thomas G. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". In: *J. Artif. Intell. Res.* 13.1 (2000), pp. 227–303.

Doner, John. "Tree acceptors and some of their applications". In: *Journal of Computer and System Sciences* 4.5 (1970), pp. 406–451.

Duret-Lutz, Alexandre et al. "Spot 2.0 - A Framework for LTL and \omega - Automata Manipulation". In: *ATVA*. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 122–129.

Efthymiadis, Kyriakos and Daniel Kudenko. "A comparison of plan-based and abstract MDP reward shaping". In: *Connect. Sci.* 26.1 (2014), pp. 85–99. DOI: 10.1080/09540091.2014.885283. URL: https://doi.org/10.1080/09540091.2014.885283,https://www.tandfonline.com/doi/full/10.1080/09540091.2014.885283.

Ehlers, R. "Symbolic Bounded Synthesis". In: *CAV*. 2010.

Ehlers, Rüdiger et al. "Supervisory control and reactive synthesis: a comparative introduction". In: *Discrete Event Dynamic Systems* 27.2 (2017), pp. 209–260.

Elgot, Calvin C. "Decision problems of finite automata design and related arithmetics". In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51.

Emerson, E. Allen. "Temporal and Modal Logic". In: *Handbook of Theoretical Computer Science*. 1990.

Fagin, Ronald et al. *Reasoning About Knowledge*. MIT Press, 1995.

Fellah, Abdelaziz, Helmut Jürgensen, and Sheng Yu. "Constructions for alternating finite automata". In: *Int. J. Comput. Math.* 35.1-4 (1990), pp. 117–132.

Felli, Paolo, Giuseppe De Giacomo, and Alessio Lomuscio. "Synthesizing Agent Protocols From LTL Specifications Against Multiple Partially-Observable Environments". In: *KR*. AAAI Press, 2012.

Fischer, Michael J. and Richard E. Ladner. "Propositional dynamic logic of regular programs". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 194–211. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(79)90046-1. URL: http://www.sciencedirect.com/science/article/pii/0022000079900461.

Fogarty, Seth et al. "Profile trees for Büchi word automata, with application to determinization". In: *Inf. Comput.* 245 (2015), pp. 136–151.

Fried, Dror, Lucas M Tabajara, and Moshe Y Vardi. "BDD-based boolean functional synthesis". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 402–421.

Gabaldon, Alfredo. "Precondition Control and the Progression Algorithm". In: *ICAPS*. AAAI, 2004, pp. 23–32.

Gabbay, D. et al. *On the Temporal Analysis of Fairness*. Tech. rep. Jerusalem, Israel, Israel, 1997.

Gaon, Maor and Ronen I. Brafman. "Reinforcement Learning with Non-Markovian Rewards". In: *AAAI*. AAAI Press, 2020, pp. 3980–3987.

Garcia, Javier and Fernando Fernandez. "A comprehensive survey on safe reinforcement learning". In: *J. Mach. Learn. Res.* 16 (2015), pp. 1437–1480.

Garey, M. R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

Geffner, Hector and Blai Bonet. *A Coincise Introduction to Models and Methods for Automated Planning*. Morgan&Claypool, 2013.

Gerevini, Alfonso et al. "Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners". In: *Artif. Intell.* 173.5-6 (2009), pp. 619–668.

Gerevini, Alfonso E et al. "Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners". In: *AIJ* 173.5-6 (2009), pp. 619–668.

Gerth, Rob et al. "Simple on-the-fly automatic verification of linear temporal logic". In: *PSTV*. Vol. 38. IFIP Conference Proceedings. Chapman & Hall, 1995, pp. 3–18.

Ghallab, M., D. S. Nau, and P. Traverso. *Automated planning - theory and practice*. 2004.

Ghallab, Malik, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

Gödel, Kurt. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für mathematik und physik* 38.1 (1931), pp. 173–198.

Goldman, R. P. and M. S. Boddy. "Expressive Planning and Explicit Knowledge". In: *AIPS*. 1996.

Gottlob, Georg. "Computer science as the continuation of logic by other means". In: *Keynote Address, European Computer Science Summit* (2009).

Gretton, Charles. "A More Expressive Behavioral Logic for Decision-Theoretic Planning". In: *PRICAI*. Vol. 8862. Lecture Notes in Computer Science. Springer, 2014, pp. 13–25.

Grounds, Matthew Jon and Daniel Kudenko. "Combining Reinforcement Learning with Symbolic Planning". In: *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning, 5th, 6th, and 7th European Symposium, ALAMAS 2005-2007 on Adaptive and Learning Agents and Multi-Agent Systems, Revised Selected Papers*. 2007, pp. 75–86. DOI: 10.1007/978-3-540-77949-0_6. URL: https://doi.org/10.1007/978-3-540-77949-0%5C_6.

Grzes, M. and D. Kudenko. "Plan-based reward shaping for reinforcement learning". In: *Proc. of the 4th International IEEE Conference on Intelligent Systems*. 2008, pp. 10–22.

Grzes, Marek. "Improving exploration in reinforcement learning through domain knowledge and parameter analysis". PhD thesis. University of York, 2010.

Grześ, Marek. "Reward Shaping in Episodic Reinforcement Learning". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '17. São Paulo, Brazil: International Foundation for Autonomous

Agents and Multiagent Systems, 2017, pp. 565–573. URL: http://dl.acm.org/citation.cfm?id=3091125.3091208.

Hadfield-Menell, Dylan et al. "The Off-Switch Game". In: *IJCAI*. 2017, pp. 220–227.

Halpern, Joseph Y. et al. "On the unusual effectiveness of logic in computer science". In: *Bull. Symb. Log.* 7.2 (2001), pp. 213–236.

Hasanbeig, Mohammadhosein, Alessandro Abate, and Daniel Kroening. "Logically-Constrained Neural Fitted Q-iteration". In: *AAMAS*. International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 2012–2014.

Hasanbeig, Mohammadhosein, Yiannis Kantaros, et al. "Reinforcement Learning for Temporal Logic Control Synthesis with Probabilistic Satisfaction Guarantees". In: *CDC*. IEEE, 2019, pp. 5338–5343.

Haslum, P. et al. *An Introduction to the Planning Domain Definition Language*. 2019.

Hengst, Bernhard. "Hierarchical Reinforcement Learning". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 495–502. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_363. URL: https://doi.org/10.1007/978-0-387-30164-8_363.

Henriksen, J. G., J. L. Jensen, et al. "Mona: Monadic Second-order Logic in Practice". In: *TACAS*. 1995.

Henriksen, Jesper G., Jakob Jensen, et al. "Mona: Monadic second-order logic in practice". In: 1995, pp. 89–110.

Heule, Marijn and Sicco Verwer. "Exact DFA Identification Using SAT Solvers". In: *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*. 2010, pp. 66–79.

Hoffmann, J. and R. I. Brafman. "Contingent Planning via Heuristic Forward Search with Implicit Belief States". In: *ICAPS*. 2005.

Hopcroft, John. "An n log n algorithm for minimizing states in a finite automaton". In: *Theory of machines and computations*. 1971, pp. 189–196.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2006.

Icarte, Rodrigo Toro, Toryn Klassen, et al. "Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning". In: *ICML*. 2018, pp. 2107–2116.

Icarte, Rodrigo Toro, Toryn Q Klassen, et al. "Teaching Multiple Tasks to an RL Agent using LTL". In: (2018).

— "Using Advice in Model-Based Reinforcement Learning". In: *The 3rd Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*. 2017.

Icarte, Rodrigo Toro, Ethan Waldie, et al. "Learning Reward Machines for Partially Observable Reinforcement Learning". In: *NIPS*. 2019, pp. 15523–15534.

Immerman, Neil. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

J. Nilsson, N. *Principles of Artificial Intelligence*. 1982.

Jiménez, P. and C. Torras. "An efficient algorithm for searching implicit AND/OR graphs with cycles". In: *Artif. Intell.* 124.1 (2000). DOI: 10.1016/S0004-3702(00)00063-1.

Jobstmann, B. and R. Bloem. "Optimizations for LTL Synthesis". In: *FMCAD*. 2006.

Jothimurugan, Kishor et al. "Compositional Reinforcement Learning from Logical Specifications". In: *NeurIPS*. 2021, pp. 10026–10039.

Kaelbling, Leslie Pack, Michael L. Littman, and Anthony R. Cassandra. "Planning and Acting in Partially Observable Stochastic Domains". In: *Artif. Intell.* 101.1-2 (1998), pp. 99–134.

Kamp, Johan Anthony Wilem. *Tense logic and the theory of linear order*. University of California, Los Angeles, 1968.

Karpathy, Andrej. *REINFORCEjs: WaterWorld demo*. https://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html. Accessed: 15-03-2020. 2015.

Khoussainov, Bakhadyr and Anil Nerode. *Automata Theory and Its Applications*. Secaucus, NJ, USA: Birkhauser Boston, Inc., 2001. ISBN: 3764342072.

Klarlund, Nils. "Mona & Fido: The logic-automaton connection in practice". In: *CSL*. 1997, pp. 311–326.

Klarlund, Nils and Anders Møller. *Mona version 1.4: User manual*. BRICS, Department of Computer Science, University of Aarhus Denmark, 2001.

Kupferman, Orna. "On High-Quality Synthesis". In: *Computer Science - Theory and Applications - 11th International Computer Science Symposium in Russia, CSR 2016, St. Petersburg, Russia, June 9-13, 2016, Proceedings*. 2016, pp. 1–15.

Kupferman, Orna and Moshe Y. Vardi. "From Linear Time to Branching Time". In: *ACM Trans. Comput. Log.* 6.2 (2005), pp. 273–294.

Lacerda, Bruno, David Parker, and Nick Hawes. "Optimal Policy Generation for Partially Satisfiable Co-Safe LTL Specifications". In: *IJCAI*. 2015, pp. 1587–1593.

Leike, Jan et al. "AI Safety Gridworlds". In: *CoRR* abs/1711.09883 (2017).

Leiss, Ernst L. "Succint Representation of Regular Languages by Boolean Automata". In: *Theor. Comput. Sci.* 13 (1981), pp. 323–330.

Leon Illanes, Le et al. "Symbolic Planning and Model-Free Reinforcement Learning: Training Taskable Agents". In: *Proc. of 4th Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*. 2019. URL: http://www.cs.toronto.edu/~lillanes/papers/IllanesYTM-rldm2019-symbolic.pdf.

Leonetti, Matteo, Luca Iocchi, and Peter Stone. "A synthesis of automated planning and reinforcement learning for efficient, robust decision-making". In: *Artificial Intelligence* 241 (2016), pp. 103–130. DOI: 10.1016/j.artint.2016.07.004. URL: https://doi.org/10.1016/j.artint.2016.07.004.

Levine, John. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

Li, J., K. Y. Rozier, et al. "SAT-Based Explicit LTL$_f$ Satisfiability Checking". In: *AAAI*. 2019.

Li, Xiao, Cristian Ioan Vasile, and Calin Belta. "Reinforcement learning with temporal logic rewards". In: *IROS*. IEEE, 2017, pp. 3834–3839.

Lichtenstein, Orna and Amir Pnueli. "Checking That Finite State Concurrent Programs Satisfy Their Linear Specification". In: *POPL*. ACM Press, 1985, pp. 97–107.

Linz, Peter. *An introduction to formal languages and automata*. Jones and Bartlett Publishers, 2006.

Littman, Michael. *The Reward Hypothesis - Markov Decision Processes*. en. URL: https://www.coursera.org/lecture/fundamentals-of-reinforcement-learning/michael-littman-the-reward-hypothesis-q6x0e (visited on 05/31/2022).

Littman, Michael L. "Programming agent via rewards". In: *Invited talk at IJCAI*. 2015.

Littman, Michael L. et al. "Environment-Independent Task Specifications via GLTL". In: *CoRR* abs/1704.04341 (2017).

Littman, Michael Lederman. "Programming agent via rewards." Invited talk at IJCAI. 2015.

Ly, Linh Thao et al. “A Framework for the Systematic Comparison and Evaluation of Compliance Monitoring Approaches”. In: *EDOC*. 2013, pp. 7–16.

Maggi, Fabrizio Maria et al. “Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata”. In: *BPM*. Vol. 6896. Lecture Notes in Computer Science. Springer, 2011, pp. 132–147.

Mahanti, A. and A. Bagchi. “AND/OR Graph Heuristic Search Methods”. In: *J. ACM* 32.1 (1985). DOI: 10.1145/2455.2459.

Manna, Zohar and Richard J. Waldinger. *The deductive foundations of computer programming - a one-volume version of "The logical basis for computer programming"*. Addison-Wesley, 1993.

Maslov, AN. “Estimates of the number of states of finite automata”. In: *Doklady Akademii Nauk*. 1970, pp. 1266–1268.

Mattmüller, R. “Informed progression search for fully observable nondeterministic planning”. PhD thesis. 2013.

Mattmüller, R. et al. “Pattern Database Heuristics for Fully Observable Nondeterministic Planning”. In: *ICAPS*. 2010.

Mealy, George H. “A method for synthesizing sequential circuits”. In: *Bell Syst.* 34 (Sept. 1955), pp. 1045–1079.

Meyer, Albert R. “Weak monadic second order theory of succesor is not elementary-recursive”. In: *Logic colloquium*. Springer. 1975, pp. 132–154.

Mnih, Volodymyr et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

Mohri, Mehryar. “Finite-State Transducers in Language and Speech Processing”. In: *Comput. Linguistics* 23.2 (1997), pp. 269–311.

Montali, Marco et al. “Declarative specification and verification of service choreographiess”. In: *ACM Trans. Web* 4.1 (2010), 3:1–3:62.

Moore, Andrew W. “Variable Resolution Dynamic Programming”. In: *ML91*. 1991, pp. 333–337.

Moore, Edward F. “Gedanken-Experiments on Sequential Machines”. In: *Automata Studies* (Dec. 1956), pp. 129–154.

“Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic”. In: *Logic, Methodology and Philosophy of Science*. Ed. by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Vol. 44. Studies in Logic and the Foundations of Mathematics. Elsevier, 1966, pp. 1–11. DOI: https://doi.org/10.1016/S0049-237X(09)70564-6. URL: https://www.sciencedirect.com/science/article/pii/S0049237X09705646.

Ng, Andrew Y., Daishi Harada, and Stuart J. Russell. “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. In: *Proceedings of the Sixteenth International Conference on Machine Learning*. ICML ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–287. ISBN: 1-55860-612-2. URL: http://dl.acm.org/citation.cfm?id=645528.657613.

Ng, Andrew Y. and Stuart J. Russell. “Algorithms for Inverse Reinforcement Learning”. In: *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*. 2000, pp. 663–670.

Nilsson, N. J. *Problem-solving methods in artificial intelligence*. 1971.

Ohrstrom, Peter and Per Hasle. *Temporal logic: From ancient ideas to artificial intelligence*. Vol. 57. Springer Science & Business Media, 2007.

OpenAI. *FrozenLake-v0*. https://gym.openai.com/envs/FrozenLake-v0/. Accessed: 30-06-2020. 2016.

Orseau, Laurent and Stuart Armstrong. "Safely Interruptible Agents". In: *UAI*. 2016.

Papadimitriou, Christos H. *Computational complexity*. Academic Internet Publ., 2007.

Patrizi, Fabio et al. "Computing Infinite Plans for LTL Goals Using a Classical Planner". In: *IJCAI*. 2011, pp. 2003–2008.

Pearl, Judea. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984.

Pesic, Maja and Wil M. P. van der Aalst. "A Declarative Approach for Flexible Business Processes Management". In: *Business Process Management Workshops*. Vol. 4103. Lecture Notes in Computer Science. Springer, 2006, pp. 169–180.

Pesic, Maja, Dragan Bosnacki, and Wil M. P. van der Aalst. "Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance". In: *SPIN*. Vol. 6349. Lecture Notes in Computer Science. Springer, 2010, pp. 146–161.

Pesic, Maja, Helen Schonenberg, and Wil M. P. van der Aalst. "DECLARE: Full Support for Loosely-Structured Processes". In: *Proc. of the 11th IEEE Int. Enterprise Distributed Object Computing Conf. (EDOC)*. IEEE Computer Society, 2007, pp. 287–300.

Pešić, Maja, Dragan Bošnački, and Wil MP van der Aalst. "Enacting declarative languages using LTL: avoiding errors and improving performance". In: *SPIN*. 2010, pp. 146–161.

Pnueli, A. and R. Rosner. "On the Synthesis of a Reactive Module". In: *POPL*. 1989.

Pnueli, Amir. "Linear and Branching Structures in the Semantics and Logics of Reactive Systems". In: *ICALP*. Vol. 194. Lecture Notes in Computer Science. Springer, 1985, pp. 15–32.

— "The Temporal Logic of Programs". In: *FOCS*. IEEE Computer Society, 1977, pp. 46–57.

Prior, Arthur N. *Time and modality*. John Locke Lecture, 2003.

Puterman, Martin L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.

Queille, Jean-Pierre and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *Symposium on Programming*. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351.

Quint, Eleanor et al. "Formal Language Constraints for Markov Decision Processes". In: *CoRR* abs/1910.01074 (2019).

Rabin, Michael O. and Dana S. Scott. "Finite Automata and Their Decision Problems". In: *IBM J. Res. Dev.* 3.2 (1959), pp. 114–125.

Raman, Vasumathi et al. "Model predictive control with signal temporal logic specifications". In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. 2014, pp. 81–87.

Ray, Alex, Joshua Achiam, and Dario Amodei. "Benchmarking safe exploration in deep reinforcement learning". In: *arXiv preprint arXiv:1910.01708* 7 (2019), p. 1.

Reif, J. H. "The Complexity of Two-Player Games of Incomplete Information". In: *JCSS* 29.2 (1984).

Reiter, Raymond. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT press, 2001.

Reynolds, John C. *Theories of programming languages*. Cambridge University Press, 1998.

Rintanen, J. "Complexity of Planning with Partial Observability". In: *ICAPS*. 2004.

Ronca, Alessandro and Giuseppe De Giacomo. "Efficient PAC Reinforcement Learning in Regular Decision Processes". In: *IJCAI*. ijcai.org, 2021, pp. 2026–2032.

Ronca, Alessandro, Gabriel Paludo Licks, and Giuseppe De Giacomo. "Markov Abstractions for PAC Reinforcement Learning in Non-Markov Decision Processes". In: (2022).

Rudell, Richard. "Dynamic variable ordering for ordered binary decision diagrams". In: *ICCAD*. IEEE, 1993, pp. 42–47.

Rummery, Gavin A and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. Citeseer, 1994.

Russell, Stuart J. and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2010.

Scutellà, M. G. "A Note on Dowling and Gallier's Top-down Algorithm for Propositional Horn Satisfiability". In: *J. Log. Program.* 8.3 (1990), pp. 265–273.

Silva, João P. Marques and Karem A. Sakallah. "GRASP - a new search algorithm for satisfiability". In: *ICCAD*. IEEE, 1996, pp. 220–227.

Silver, David et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–359.

Simsek, Özgür and Andrew G. Barto. "An intrinsic reward mechanism for efficient exploration". In: *ICML*. Vol. 148. ACM International Conference Proceeding Series. ACM, 2006, pp. 833–840.

Singh, Satinder P. and Richard S. Sutton. "Reinforcement Learning with Replacing Eligibility Traces". In: *Mach. Learn.* 22.1-3 (Jan. 1996), pp. 123–158. ISSN: 0885-6125. DOI: 10.1007/BF00114726. URL: http://dx.doi.org/10.1007/BF00114726.

Sistla, A. P. and E. M. Clarke. "The Complexity of Propositional Linear Temporal Logics". In: *J. ACM* 32.3 (July 1985), pp. 733–749. ISSN: 0004-5411. DOI: 10.1145/3828.3837. URL: http://doi.acm.org/10.1145/3828.3837.

Sohrabi, Shirin, Jorge A. Baier, and Sheila A. McIlraith. "Preferred Explanations: Theory and Generation via Planning". In: *AAAI*. 2011.

Somenzi, Fabio. "CUDD: CU Decision Diagram Package, 3.0." In: *University of Colorado at Boulder* (2015).

Sutton, Richard S. "Generalization in reinforcement learning: Successful examples using sparse coarse coding". In: *Advances in neural information processing systems* 8 (1995).

— "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming". In: *Machine Learning, Proceedings of the Seventh International Conference on Machine Learning, Austin, Texas, USA, June 21-23, 1990*. 1990, pp. 216–224. DOI: 10.1016/b978-1-55860-141-3.50030-4. URL: https://doi.org/10.1016/b978-1-55860-141-3.50030-4.

— "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3.1 (Aug. 1988), pp. 9–44. ISSN: 1573-0565. DOI: 10.1007/BF00115009. URL: https://doi.org/10.1007/BF00115009.

Sutton, Richard S. and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

Sutton, Richard S., Doina Precup, and Satinder P. Singh. "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning". In: *Artif. Intell.* 112.1-2 (1999), pp. 181–211. DOI: 10.1016/S0004-3702(99)00052-1. URL: https://doi.org/10.1016/S0004-3702(99)00052-1.

Tabajara, L. M. and M. Y. Vardi. "Partitioning Techniques in LTL$_f$ Synthesis". In: *IJCAI*. 2019.

Tabajara, Lucas Martinelli and Moshe Y Vardi. "Partitioning Techniques in LTLf Synthesis." In: *IJCAI*. 2019, pp. 5599–5606.

Tabakov, Deian and Moshe Y Vardi. "Experimental evaluation of classical automata constructions". In: *LPAR*. 2005, pp. 396–411.

Tamm, Hellis and Margus Veanes. "Theoretical Aspects of Symbolic Automata". In: *SOFSEM*. Vol. 10706. Lecture Notes in Computer Science. Springer, 2018, pp. 428–441.

Tarski, Alfred. "Der Wahrheitsbegriff in den formalisierten Sprachen". In: *Studia philosophica* 1 (1936).

Thanh To, S., E. Pontelli, and T. Cao Son. "A Conformant Planner with Explicit Disjunctive Representation of Belief States". In: *ICAPS*. 2009.

Thatcher, James W. and Jesse B. Wright. "Generalized finite automata theory with an application to a decision problem of second-order logic". In: *Mathematical systems theory* 2.1 (1968), pp. 57–81.

Thiébaux, Sylvie et al. "Decision-Theoretic Planning with non-Markovian Rewards". In: *J. Artif. Intell. Res.* 25 (2006), pp. 17–74.

Thomas, Wolfgang. "Star-free regular sets of ω-sequences". In: *Information and Control* 42.2 (1979), pp. 148–156. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(79)90629-6. URL: http://www.sciencedirect.com/science/article/pii/S0019995879906296.

Torres, Jorge and Jorge A. Baier. "Polynomial-Time Reformulations of LTL Temporally Extended Goals into Final-State Goals". In: *IJCAI*. 2015.

Trakhtenbrot, B.A. "Finite automata and the logic of single-place predicates." English. In: *Sov. Phys., Dokl.* 6 (1961), pp. 753–755. ISSN: 0038-5689.

Tseitin, Grigori S. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

Turing, Alan Mathison et al. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

Vaandrager, Frits W. "Model learning". In: *Commun. ACM* 60.2 (2017), pp. 86–95.

Vardi, Moshe Y. "Logic and automata: A match made in heaven". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2003, pp. 64–65.

— "An Automata-Theoretic Approach to Linear Temporal Logic". In: *Banff Higher Order Workshop*. Vol. 1043. Lecture Notes in Computer Science. Springer, 1995, pp. 238–266.

— "From Church and Prior to PSL". In: *25 Years of Model Checking - History, Achievements, Perspectives*. 2008, pp. 150–171.

Vardi, Moshe Y. and Pierre Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In: *LICS*. IEEE Computer Society, 1986, pp. 332–344.

— "Reasoning About Infinite Computations". In: *Inf. Comput.* 115.1 (1994), pp. 1–37.

Veanes, Margus, Nikolaj S. Bjørner, and Leonardo Mendonça de Moura. "Symbolic Automata Constraint Solving". In: *LPAR (Yogyakarta)*. Vol. 6397. Lecture Notes in Computer Science. Springer, 2010, pp. 640–654.

Veanes, Margus, Peli De Halleux, and Nikolai Tillmann. "Rex: Symbolic regular expression explorer". In: *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE. 2010, pp. 498–507.

Watkins, Christopher J. C. H. and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

Watkins, Christopher John Cornish Hellaby. "Learning from delayed rewards". PhD thesis. 1989.

Wen, Min, Rüdiger Ehlers, and Ufuk Topcu. "Correct-by-synthesis reinforcement learning with temporal logic constraints". In: *IROS*. 2015, pp. 4983–4990.

Whitehead, Steven D. and Long-Ji Lin. "Reinforcement learning of non-Markov decision processes". In: *Artificial Intelligence* 73.1 (1995). Computational Research on Interaction and Agency, Part 2, pp. 271–306. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(94)00012-P. URL: http://www.sciencedirect.com/science/article/pii/000437029400012P.

Wolper, Pierre. "Temporal logic can be more expressive". In: *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)* (1981), pp. 340–348.

Xiao, S. et al. "On-the-fly Synthesis for LTL over Finite Traces". In: *AAAI*. 2021.

Yang, Fangkai et al. "PEORL: Integrating Symbolic Planning and Hierarchical Reinforcement Learning for Robust Decision-Making". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 2018, pp. 4860–4866. DOI: 10.24963/ijcai.2018/675. URL: https://doi.org/10.24963/ijcai.2018/675.

Yu, Fang, Muath Alkhalaf, and Tevfik Bultan. "Stranger: An Automata-Based String Analysis Tool for PHP". In: *TACAS*. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 154–157.

Yu, Fang, Tevfik Bultan, et al. "Symbolic string verification: An automata-based approach". In: *SPIN*. 2008, pp. 306–324.

Yu, Sheng, Qingyu Zhuang, and Kai Salomaa. "The state complexities of some basic operations on regular languages". In: *Theoretical Computer Science* 125.2 (1994), pp. 315–328.

Zhu, S., G. De Giacomo, et al. "LTL$_f$ Synthesis with Fairness and Stability Assumptions". In: *AAAI*. 2020.

Zhu, S., G. Pu, and M. Y. Vardi. "First-Order vs. Second-Order Encodings for LTL$_f$-to-Automata Translation". In: *TAMC*. 2019.

Zhu, Shufang, Lucas M. Tabajara, Jianwen Li, et al. "Symbolic LTLf Synthesis". In: *IJCAI*. 2017, pp. 1362–1369.

Zhu, Shufang, Lucas M. Tabajara, Geguang Pu, et al. "On the Power of Automata Minimization in Temporal Synthesis". In: *GandALF*. Vol. 346. EPTCS. 2021, pp. 117–134.